More Academy

Fundamentos de Python ¶

Tópicos:

1. Primeiros passos!
2. Variáveis
3. Números
4. Strings
5. Operadores
6. Estruturas condicionais
7. Estruturas de repetição

1. Primeiros passos!

A primeira coisa que alguém precisa ter em mente quando adentra no mundo da programação é o que são variáveis. De forma geral, uma variável é onde armazenamos alguma informação. Não é a intenção se alongar sobre assuntos técnicos, como, por exemplo, a forma que o computador armazena essas informações na memória, contudo, você precisará ter algum conhecimento básico sobre alguns tipos de variáveis e de como a linguagem reagirá a ela.

No geral, a maioria das linguagens, e também o Python, utiliza majoritariamente 4 tipos de variáveis. São eles:

- String (caracteres alfanúmericos)
- Integer (Números Inteiros)
- Float (Números Fracionados)
- Boolean (Booleano)

Outra coisa que precisa ter em mente é que o Python, pelo seu próprio interpretador, ele "define" um tipo de variável automáticamente, assim como ele já possui uma forma de demonstrar os valores contidos na variável quando solicitado.

Para atribuir um valor a uma variável, basta escrever o nome da variável, que pode ser qualquer sequência de letras e números e, em seguida, adicionar o sinal de igualdade, seguido do valor que deseja atribuir a variável. Veja como é feito para atribuir os tipos básicos mencionados acima em sequência.

In [1]:

```
string = "Isso é uma string"
string
```

Out[1]:

```
'Isso é uma string'
```

In [2]:

```
integer = 2021
integer
```

Out[2]:

```
2021
```

No caso do "float", esse nome é reservado para funcionalidades do próprio Python, assim como "str" para strings, "int" para integer e "bool" para booleanos.

Outro ponto importante é que, você deve utilizar pontos em vez de vírgulas para indicar a fração. O uso da vírgula no Python é reservado para diferenciar uma variável da outra, por exemplo. Darei um exemplo mais abaixo.

In [3]:

```
fracionado = 20.21
fracionado
```

Out[3]:

```
20.21
```

In [4]:

```
boolean = True
boolean
```

Out[4]:

```
True
```

Uma funcionalidade que o Python prove na hora da declaração de variáveis é a de realizar uma declaração de variáveis aninhadas. Pode ser que agora não pareça ter uma utilidade, mas conforme a complexidade do seu código aumenta é bom conhecer certos "atalhos".

No exemplo abaixo, você declara que a variável "a" receberá o valor 1, enquanto a variável "b" receberá o valor 2.50. Veja nesse caso como a vírgula foi utilizada.

In [5]:

```
a, b = 1 , 2.50

a, b
```

Out[5]:

```
(1, 2.5)
```

Você poderá declarar variáveis dessa forma também, ou seja, pedindo para que o valor de "c" seja o valor de "d" e que o valor de "d" seja 2. Por consequência, o valor de ambas serão o mesmo.

In [6]:

```
c = d = 2

c,d
```

Out[6]:

```
(2, 2)
```

Para fins da proposta deste curso, fiquemos com esses 4 tipos por enquanto.

1. Exibição de dados

Uma das funcionalidades que mais importam numa linguagem de programação é a capacidade de imprimir dados na tela. Para isso, o Python conta com a função "print()". A função print() possui algumas funcionalidades importantes que podem auxilizar na hora da programação e facilitar a vida do programador.

Caso tenha notado, e se questionado, quando declaramos as variáveis, não foi preciso escrever a função print(), contudo, ele pede para que seja exibido somente o valor contido apenas na variável, caso queira exibir algo além, será necessário utilizar uma função como print().

In [7]:

```
print("Olá, mundo!") # pede para exibir a mensagem sem que ela seja salva em uma variável
```

```
Olá, mundo!
```

In [8]:

```
print(a) # pede para exibir aquilo que está armazenado na variável "a"
```

```
1
```

Agora, vamos montar um texto genérico para trabalharmos com a exibição desses dados em meio a outras frases.

Basicamente, há duas formas de se fazer isso, digamos que, uma é da forma "correta" e outra da forma "errada", mas ambas podem ser utilizadas.

Nesse caso, pode-se dizer que é da maneira incorreta, pois, eventualmente, você terá que "converter" o tipo de dado para que ele possa ser exibido.

In [9]:

```
nome = "Carlos"
idade = "18"
peso = "67.5"
print("Olá, "+nome+" !")
print("Sua idade é "+idade+" anos.")
print("Seu peso é "+peso+" kg.")
```

```
Olá, Carlos !
Sua idade é 18 anos.
Seu peso é 67.5 kg.
```

Já esse, é a maneira mais correta de se exibir um dado.

In [10]:

```
print(f"Olá, {nome} !")
print(f"Sua idade é {idade} anos.")
print(f"Seu peso é {peso} kg.")
```

```
Olá, Carlos !
Sua idade é 18 anos.
Seu peso é 67.5 kg.
```

Há, ainda, uma terceira forma de exibir uma mensagem, porém, é um pouco desconhecida. Nesse caso, você poderá fazer da seguinte forma:

In [11]:

```
print("""
    Uma forma, um tanto quanto
    diferente de realizar o
    print, na linguagem
    Python
    """)
```

```
    Uma forma, um tanto quanto
    diferente de realizar o
    print, na linguagem
    Python
```

Ademais, para exibir os dados armazenados numa variável segue da mesma maneira. Inclua um "f" no ínicio e o a variável entre chaves.

Outra ferramenta bem útil enquanto está aprendendo Python, é saber inserir dados enquanto se trabalha em um "console". Contudo, devo lembrar que, quanto mais complexo for a ideia, alguns recursos são abandonados e outros assumidos. Mas, para tópicos introdutórios, é importante saber alguns truques e saber que eles são possíveis de serem feitos.

Para isso, é preciso utilizar a função "input()" para solicitar que um dado seja digitado. Veja nos exemplos a seguir.

In [12]:

```
nome = input("Digite seu nome: ")
nome
```

Out[12]:

```
'More'
```

In [13]:

```
idade = input("Digite sua idade: ")
peso = input("Digite seu peso: ")
```

In [14]:

```
print(f"""
    Olá, {nome}.
    Você tem {idade} anos
    e {peso} quilogramas.
    """)
```

```
    Olá, More.
    Você tem 21 anos
    e 77 quilogramas.
```

3. Números

Os tipos númericos são extremamente importantes na hora de utilizar qualquer linguagem. Veja que é possível fazer com alguns desses dados.

In [15]:

```
x = 10  #int
y = 10.5  #float
z = 10+1j  #complex
```

Veja, nesse caso, como cada um dos dados são apresentados.

In [16]:

```
print(type(x)) # int
print(type(y)) # float
print(type(z)) # complex
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

Lá em cima, falei que pode ocorrer alguns erros quando optamos por algumas formas incorretas na utilização dos dados. Uma delas é quando você tenta fazer operações matemáticas com o tipo de dado errado.

Isso é, se você tem dois números inteiros e realizar uma operação matemática, por exemplo 1 + 2, você terá o retorno da operação matemática, que é 3. Mas se você possuir um número inteiro e uma string, muito provavelmente você não conseguirá realizar nenhuma operação matemática, mas realizará uma concatenação, ou retornará um erro. O que pode causar surpresa na hora de rodar o código.

In [17]:

```
'1' + '5' # veja que, nesse caso, retornará uma concatenação
          # caso seja dois tipos de dados, poderá retornar um erro
```

Out[17]:

```
'15'
```

Veja, no caso abaixo, que a função input() retornará uma string.

In [18]:

```
idade = input("Digite sua idade: ")
type(idade)
```

Out[18]:

```
str
```

Caso queira que os dados retornem no tipo desejado, você poderá usar "funções" pontuais para converter esses dados. Nesse caso, poderá ser utilizado a função int(), para converter em inteiro, float(), para converter em "flutuantes", ou str(), para converter para strings.

Veja que, abaixo, é utilizado a função para converter um string para float, por exemplo.

In [19]:

```
idade = float(idade)
tipo = type(idade)

print(f"""
    Veja que o valor da variável idade é {idade}.
    E que seu tipo será {tipo}
    """)
```

```
    Veja que o valor da variável idade é 22.0.
    E que seu tipo será <class 'float'>
```

Como dito anteriormente, é possível realizar diversas operações matemáticas com o Python. Aqui abaixo, está alguns exemplos para ver a capacidade.

In [20]:

```
10.5 + 10 # soma
```

Out[20]:

```
20.5
```

In [21]:

```
10.5 - 10 # substração
```

Out[21]:

```
0.5
```

In [22]:

```
10/5 # divisão
```

Out[22]:

```
2.0
```

In [23]:

```
5*5 # multiplicação
```

Out[23]:

```
25
```

In [24]:

```
7%2 # resto da divisão
```

Out[24]:

```
1
```

In [25]:

```
2**10 # potência
```

Out[25]:

```
1024
```

4. Strings

Em algumas situações, pode ser necessário trabalhar com strings. Para isso, o Python possui algumas funcionalidades já prontas para isso.

In [26]:

```
nome = "UserPython"
```

É possível que você trabalhe com strings utilizando a posição onde cada palavra ocupa. Lembre-se que, no Python, a primeira posição é zero. Sendo assim, veja os exemplos a seguir.

In [27]:

```
nome[0] # primeira letra
```

Out[27]:

```
'U'
```

In [28]:

```
nome[4] # quinta letra
```

Out[28]:

```
'P'
```

In [29]:

```
nome[0:4] # da primeira até antes da quinta letra
```

Out[29]:

```
'User'
```

In [30]:

```
nome[4:] # da quinta letra em diante
```

Out[30]:

```
'Python'
```

In [31]:

```
nome[:4] # tudo que houver antes da quinta letra
```

Out[31]:

```
'User'
```

In [32]:

```
nome[:2] # do ínicio até a antes da terceira posição
```

Out[32]:

```
'Us'
```

In [33]:

```
nome[:] # a string toda
```

Out[33]:

```
'UserPython'
```

Caso queira saber quantos caracteres existe numa frase ou variável que contenha uma string, poderá utilizar a função len().

In [34]:

```
len("Python")
```

Out[34]:

```
6
```

In [35]:

```
len(nome)
```

Out[35]:

```
10
```

Verificar se existe a palavra "Python World" dentro da frase "JavaScript". Nesse caso, o resultado será negativo. Mas veja o exemplo seguinte.

In [36]:

```
a = "Python World"
"JavaScript" in a
```

Out[36]:

```
False
```

In [37]:

```
nome in "UserPython UserPython"
```

Out[37]:

```
True
```

Outras funções prontas no Python.

In [38]:

```
nome = "joão carlso da silva"
nome
```

Out[38]:

```
'joão carlso da silva'
```

In [39]:

```
nome.lower() # deixa todos os caracteres em minúsculo
```

Out[39]:

```
'joão carlso da silva'
```

In [40]:

```
nome.upper() # deixa todos os caracteres em maiúsculo
```

Out[40]:

```
'JOÃO CARLSO DA SILVA'
```

In [41]:

```
nome.title() # deixa apenas a primeira letra de cada palavra em maiúscula
```

Out[41]:

```
'João Carlso Da Silva'
```

In [42]:

```
nome.index('a') # indica onde está a letra procurada
                # no caso, perceba que ã é diferente de a
```

Out[42]:

```
6
```

In [43]:

```
ling = "   Isso é Python!     "
ling
```

Out[43]:

```
'   Isso é Python!     '
```

In [44]:

```
ling.strip() # remove os espaços desnecessários antes e após as frases
```

Out[44]:

```
'Isso é Python!'
```

5. Operadores

Assim como existe os operadores aritméticos, existe outros operadores que o Python poderá trabalhar. Nessa seção, darei 3 exemplos de operadores que podem ser utilizados.

5.1 Operadores de atribuição

São aqueles que atribui um valor a uma variável. Veja alguns exemplos abaixo e como eles podem facilitar a sua vida.

Digamos que você tenha uma variável qualquer com um valor inteiro e deseja aplicar uma operação matemática a essa variável. Poderá ser feita da seguinte forma.

In [45]:

```
x = 100
```

In [46]:

```
x += 100 # somará 100 a variável x
x
```

Out[46]:

```
200
```

In [47]:

```
x -= 100 # removerá 100 da variável x
x
```

Out[47]:

```
100
```

In [48]:

```
x *= 100 # multiplicará a variável x por 100
x
```

Out[48]:

```
10000
```

In [49]:

```
x /= 100 # divide a variável x por 100
         # perceba que, nesse caso,
         # o próprio python realizou a conversão da variável para float
x
```

Out[49]:

```
100.0
```

In [50]:

```
x %= 100 # retorna o restante da divisão de x por 100
x
```

Out[50]:

```
0.0
```

5.2 Operadores de comparação

São aqueles que comparam um valor ou variável com outro valor ou outra variável e retorna um resultado booleano com verdadeiro ou falso.

In [51]:

```
10 == 10 # valor 'a' é igual do valor 'b'
```

Out[51]:

```
True
```

In [52]:

```
10 != 20 # valor 'a' é diferente do valor 'b'
```

Out[52]:

```
True
```

In [53]:

```
10 >= 5 # valor 'a' maior, ou igual, ao valor 'b'
```

Out[53]:

```
True
```

In [54]:

```
10 > 5 # valor 'a' é maior que o valor 'b'
```

Out[54]:

```
True
```

In [55]:

```
10 <= 5 # valor 'a' menor, ou igual, ao valor 'b'
```

Out[55]:

```
False
```

In [56]:

```
10 < 5  # valor 'a' é menor que o valor 'b'
```

Out[56]:

```
False
```

5.3 Operadores lógicos

Os operadores lógicos podem ser conhecidos também por "circuitos digitais" e possuem as mesmas aplicações, caso você já os tenha aprendido. Eles funcionarão baseado nas expressões de "verdadeiro" e "falso" que são recebidas e comparadas com outros "operadores" que também retornam verdadeiro ou falso.

Não é a função desse curso, mas é possível trabalhar com os circuitos digitais NOT (como negação), AND, OR, NAND e NOR,

No Python, os operadores AND e OR são as próprias palavras, para aplicar a negação, inclua a palavra NOT após o operador.

In [57]:

```
x = 10
```

In [58]:

```
(x<20) and (x>2)  # operador AND
                  # retorna True se ambas as declarações forem verdadeiras
```

Out[58]:

```
True
```

In [59]:

```
(x<20) and not (x>2) # opererador NAND
                     # retornará False se ambas as declarações forem verdadeiras
```

Out[59]:

```
False
```

In [60]:

```
(x!=10) or (x>2) # operador OR
                 # retornará True se pelo menos uma das declarações forem verdad
eiras
```

Out[60]:

```
True
```

In [61]:

```
(x!=10) or not (x>2) # operador NOR
                     # retornará False se pelo menos uma das declarações forem f
alsas
```

Out[61]:

```
False
```

Veja exemplos abaixo de como os operadores funcionam nas operações básicas. Faça o teste, por si mesmo, incluindo os operadores negativos para ver como sairá a resposta.

In [62]:

```
True and True
```

Out[62]:

```
True
```

In [63]:

```
True and False
```

Out[63]:

```
False
```

In [64]:

```
False and True
```

Out[64]:

False

In [65]:

```
False and False
```

Out[65]:

False

Tabela 'and'

- VV - V
- VF - F
- FV - F
- FF - F

In [66]:

```
True or True
```

Out[66]:

True

In [67]:

```
True or False
```

Out[67]:

True

In [68]:

```
False or True
```

Out[68]:

True

In [69]:

```
False or False
```

Out[69]:

False

Tabela 'or'

- VV - V
- VF - V
- FV - V
- FF - F

Também é possível aninhar mais do que duas declarações, veja um exemplo abaixo.

In [70]:

```
True or False and True
```

Out[70]:

```
True
```

Caso se pergunte de uma aplicação mais prática da utilização desses operadores, veja que será possível obter alguns resultados com algumas linhas de código. Recomendo, com sinceridade que "perca" um tempo aprendendo mais sobre operadores, pois eles serão muito utilizados futuramente.

In [71]:

```
## declaração de duas variáveis

string1 = "Python"
string2 = "JavaScript"

## armazenamento do resultado de uma comparação entre as duas string

resultado = len(string1)<len(string2)

## retorno do resultado

if(resultado == True):
    print(f"A palavra {string1} é menor que a palavra {string2}")
else:
    print(f"A palavra {string1} é maior que a palavra {string2}")
```

```
A palavra Python é menor que a palavra JavaScript
```

9. Controle de Fluxo

Existe, basicamente, dois tipos de controle de fluxos na programação. Um para tomada de decisões e outro para repetir uma parte do código. Qual decisão tomar ou até que ponto repetir o código será decidido por meio de uma condição previamente estabelecida.

9.1 Controle de fluxo por decisão (if/else)

Muitos aspectos da programação imitam a vida. Dessa forma, algumas vezes, precisamos tomar uma decisão baseado em premissas anteriores. Por exemplo, quando vamos comprar um determinado item, antes de comprá-lo, consideramos algumas condições, ou pelo menos, deveriamos considerar, ter mais dinheiro em conta do que o valor do produto e utilidade do produto.

Ao seguirmos as duas condições previamente dadas (ter mais dinheiro em conta do que o valor do produto ou utilidade do produto), podemos formular dois caminhos. O primeiro de comprar caso os pré requisitos sejam atendimentos e o de declinar a compra caso os pré requisitos não sejam atingidos.

Em programação, a formulação dessa lógica seguiria da seguinte forma:

if condicao == True: # execute esse bloco caso verdadeiro else: # execute esse bloco caso falso

Antes de prosseguirmos, há alguns lembretes a serem feitos.

O primeiro é que, após a declaração da condição "if condicao == True" usa-se dois pontos (:). O mesmo se repete na condição "else". É importante colocar os dois pontos para que a linguagem identifique que, após tal declaração, existe uma parte do código que possa ser executada caso a condição seja atingida.

O segundo ponto é que, para você executar um código onde a condição é cumprida, você precisará identar, ou seja, deixar claro no código, que aquele trecho é um bloco a mais e que ele pertence à declaração anterior.

E, por último, e não menos importante, o if sempre, caso não seja especificado, ele buscará o valor verdadeiro, então indicar que você busca o valor "verdadeiro" é opcional. O mesmo acontece com a necessidade de ter a declaração "else", onde não é necessário a sua presença para o andamento do código. Tê-los ou não, não influenciará no andamento do código, contudo, lembre-se da filosofia do python, caso não lembre-se, execute o seguinte código abaixo "import this".

No segundo parágrago, é dito que o "Explícito é melhor que o implícito". Então, tenha sempre em mente essa filosofia. Tanto para você, quanto para o seus colegas que poderão trabalhar no mesmo código.

In [72]:

```
# Caso queira ver a filosofia do python, descomente o "import this".

#import this
```

Agora, alguns exemplos de como o código funciona

In [73]:

```
if True:
    print('Esse código será executado, pois, como dito anteriormente,')
    print('o \"if\" sempre buscará executar o código caso seja verdadeiro')
```

```
Esse código será executado, pois, como dito anteriormente,
o "if" sempre buscará executar o código caso seja verdadeiro
```

Várias condições podem ser usadas, tanto expressões matemáticas, quanto convalidação de caractéres ou testes booleanos.

In [74]:

```
if 2 < 3:
    print("Esse código será executado, pois 2 é, de fato, menor que 3")
```

```
Esse código será executado, pois 2 é, de fato, menor que 3
```

In [75]:

```
if "a" == "a":
    print("Esse código será executado, pois a letra \"a\" é, de fato, igual a le
tra \"a\"")
```

```
Esse código será executado, pois a letra "a" é, de fato, igual a le
tra "a"
```

In [76]:

```
validacao = 1

if validacao == True:
    print("Lembre-se que o 0 e 1, podem ser usados como expressões booleanas de
 Falso, ou Verdadeiros, respectivamente.")
    print("Portanto, quando construir o seu código, a depender do valor que será
validado, tenha isso em mente.")
```

```
Lembre-se que o 0 e 1, podem ser usados como expressões booleanas d
e Falso, ou Verdadeiros, respectivamente.
Portanto, quando construir o seu código, a depender do valor que se
rá validado, tenha isso em mente.
```

Agora é hora de usar alguns exemplos de como esse código se relaciona com a condição "else".

In [77]:

```
if True == False:
    print ('Esse é o código que será executado caso a declaração seja verdadeir
a')

else:
    print("Esse é o código executado caso a declaração seja Falsa")
```

```
Esse é o código executado caso a declaração seja Falsa
```

In [78]:

```
if (2 > 3) == True:
    print("Dois é menor que três")
else:
    print("Dois não é maior que três")
```

```
Dois não é maior que três
```

Outra forma de utilizar as validações na declaração, é usar o "is" em vez do sinal de comparação. Veja o mesmo código acima feito de forma diferente.

In [79]:

```
if (2 > 3) is True:
    print("Dois é menor que três")
else:
    print("Dois não é maior que três")
```

```
Dois não é maior que três
```

Outra opção que poderá ser feita, quando a decisão é só entre um e outro, é o que chamam de "operador ternário". Ele faz o mesmo que os códigos acima, porém, em uma única linha.

É uma opção útil em alguns casos, mas lembre-se que "explícito é melhor que implícito", então, não abuse deste recurso. Veja o código acima sendo reproduzido em um operador ternário.

In [80]:

```
print("Dois é menor que três" if 2>3 else "Dois é menor que três")
```

```
Dois é menor que três
```

Caso não tenha entendido, o operador funciona com a seguinte sintaxe:

se_teste_for_verdadeiro if (condicao) else se_teste_for_falso

Esse é um código útil se você precisará comparar valores booleanos rápidos e que a resposta do bloco a ser executado não for de grande complexidade. Contudo, é um trecho que tende a diminuir a eficiência do código, assim como é ilegível a depender de como o código for arquitetado. Então, de novo, use-o com cuidado.

Ainda há a ideia de que, uma condição, pode ser aplicada em mais do que dois casos. Por exemplo, digamos que precise fazer uma classifcação de "ótimo", "regular" e "abaixo da média" baseado numa pontuação. Você poderá usar a condição "elif" do Python para criar esse "meio termo.

In [81]:

```
nota = 5

if nota >= 7:
    print('ótimo')
elif nota >= 5:
    print("regular")
else:
    print("abaixo da média")
```

```
regular
```

Você poderá fazer diversas combinações a depender da situação ao qual estiver. Outra opção que é presente no Python, mas que não foi explicitamente falado, é que é possível realizar um aninhamento de condições. Lembra da história inicial, sobre ter saldo em conta "e" o produto a ser comprado ser necessário? Irei demonstrar alguns exemplos dessa condição.

Seguindo o último exemplo, digamos que você queira apenas as notas que seja 5 <= x <= 7 (o valor desejado ser maior, ou igual a cinco E menor, ou igual a sete), o que, cairia na classificação de "regular". Para fazer essa operação, o código poderia ser da seguinte forma:

In [82]:

```
nota = 5

if 5 <= nota and  nota <= 7:
    print("Essa é a categoria \"regular\".")
else:
    print("Não está na faixa de nota que desejamos")
```

```
Essa é a categoria "regular".
```

Operadores lógicos são úteis em diversas situações e uma das ferramentas mais utilizadas na programação. Vale a pena revisá-los, ou aprendê-los, caso não tenha a familiaridade com este tópico.

Desafio 01

O primeiro desafio será fazer, de forma lógica e funcional, um código que decida se compraremos, ou não, o produto nas seguintes condições:

Após a compra do produto, o saldo na conta bancária deverá ser superior a R$ 200,00. Caso ambas as condições não forem cumpridas, o produto não será comprado.

O valor que teremos na conta bancária fica a seu critério. Manipule, e crie, as variáveis para testar as diversas posições e, assim, testar se seu código realmente funciona para todas as situações.

In [83]:

```
## Código do desafio 01
```

Desafio 02

Agora, lembra do operador ternário? Ele é aplicado no python de diversas formas, mesmo que não seja de forma explícita. Por exemplo:

In [84]:

```
# 2 é menor que 3?
2**3 == 8
```

Out[84]:

```
True
```

Na expressão acima, sabemos que o código retornará Verdadeiro. Mas, pare e pense como a validação acima foi feita e tente reproduzir no espaço abaixo, seguindo uma declaração de decisão (if...else).

In [85]:

```
## Código do desafio 02
```

9.2 Controle de fluxo por repetição (for, while)

Como dito anteriormente, as vezes, você precisa repetir uma parte do código até atingir uma certa condição. Para fins de economizar tempo programando, existe duas ferramentas para controle de fluxo que podem ser extremamente úteis.

Uma é usada quando o alcance das repetições são bem conhecidos, o outra, é usada para quando a repetição deve parar após cumprir determinado objetivo.

Usarei a tabuada, uma "repetição" muito bem conhecida por todos, para exemplificiar como funciona os dois laços de repetição e qual a diferença de um para o outro.

Primeiro, farei com o laço "for" e depois com o laço "while".

In [86]:

```
#tabuada do 2 com laço de repetição for

for i in range(1,11):
    print(f"2*{i} = {2*i}")
```

```
2*1 = 2
2*2 = 4
2*3 = 6
2*4 = 8
2*5 = 10
2*6 = 12
2*7 = 14
2*8 = 16
2*9 = 18
2*10 = 20
```

In [87]:

```
#tabuada do 2 com laço de repetição while

i = 1
while i < 11:
    print(f"2*{i} = {2*i}")
    i = i + 1
```

```
2*1 = 2
2*2 = 4
2*3 = 6
2*4 = 8
2*5 = 10
2*6 = 12
2*7 = 14
2*8 = 16
2*9 = 18
2*10 = 20
```

A diferença básica de um para o outro é que, o for é bem definido, enquanto o while fará a repetição até que sua condição mude de status. O while trabalhará muito bem com valores booleanos como condições de verificação, contudo, tome cuidado ao estipular condições, pois, pode ser que o laço simplesmente não pare de se repetir. Por exemplo, digamos que você coloque algo como "while 1<2", essa condição sempre será verdadeira, portanto, o laço nunca terá fim.

9.2.1 Trabalhando com o FOR em específico

Agora, digamos que você queira saber em qual "categoria" cada uma das notas se encaixa. Veja o laço "for" abaixo.

In [88]:

```
# fazer for para mostrar o campo de notas "ótimo", "regular" e "abaixo da média"

for i in range(0,11):
    if i >= 7:
        print(f"A nota {i} é ótima.")
    elif i >= 5:
        print(f"A nota {i} é regular")
    else:
        print(f"A nota {i} é abaixo da média")
```

```
A nota 0 é abaixo da média
A nota 1 é abaixo da média
A nota 2 é abaixo da média
A nota 3 é abaixo da média
A nota 4 é abaixo da média
A nota 5 é regular
A nota 6 é regular
A nota 7 é ótima.
A nota 8 é ótima.
A nota 9 é ótima.
A nota 10 é ótima.
```

Temos uma aplicação muito útil quando você deseja aplicar alguma função em cada item ou conjunto de dados. Para isso, utilizamos uma forma chamada de "foreach" que pode ser traduzida como "para cada item". É bem intuitivo. Digamos que deseje imprimir cada um dos itens que estão dentro da lista a seguir. O "foreach" poderá ser muito útil.

In [89]:

```
# foreach

anos = [2010,
        2014,
        2018,
        2022]

for ano in anos:
    print(ano)
```

```
2010
2014
2018
2022
```

Outra forma de realizar uma interação entre os dados é utilizar a função enumerate que trabalhará com um contador. Veja que passamos duas listas, nesse caso, a cada volta, ele irá modificar o contador ou index para o seguinte, assim, alinhando um dado com o outro.

In [90]:

```
# acesso por index

trimestres = ['primeiro',
              'segundo',
              'terceiro',
              'quarto']

pibs = [123.321,
       234.432,
       345.543,
       456.654]

for index, pib in enumerate(pibs):

    trimestre = trimestres[index]
    print(f"o PIB do {trimestre} trimestre foi de {pib}")
```

```
o PIB do primeiro trimestre foi de 123.321
o PIB do segundo trimestre foi de 234.432
o PIB do terceiro trimestre foi de 345.543
o PIB do quarto trimestre foi de 456.654
```

9.2.1 Trabalhando com o WHILE em específico

Como dito anteriormente, o while é muito utilizado quando você não tem certeza de quantos loops você terá que percorrer, então, você trabalha com valores booleanos em vez de uma quantidade pré definida de laços.

In [91]:

```
var = True
i = 0

while var == True:

    print(f"O valor é {i}.")
    i += 1

    if i > 9:
        print("O valor é maior ou igual a nove")
        var = False
```

```
O valor é 0.
O valor é 1.
O valor é 2.
O valor é 3.
O valor é 4.
O valor é 5.
O valor é 6.
O valor é 7.
O valor é 8.
O valor é 9.
O valor é maior ou igual a nove
```

Lembrar de resetar a variável contadora, por "costume", no Python se usa a variáve "i". Caso não seja resetado (i = 0, por exemplo, o código poderá apresentar problemas pois o valor da variável já estará alterado.

Uma outra possibilidade é de poder usar o retorno while-else no laço while. Dessa forma, você poderá ser "informado" quando o laço se encerra e poderá prosseguir com outra operação. Por exemplo:

In [92]:

```
numero = 0
while numero <= 10:
    print(numero)
    numero += 1
else:
    print('\nO laço se encerrou')
```

```
0
1
2
3
4
5
6
7
8
9
10

O laço se encerrou
```

Estrutura de Dados

Vamos aprender um pouco sobre estruturas de dados do Python, focaremos em listas e tuplas.

Listas

As listas podem ser utilizadas para armazenar valores. Esses valores podem ser strings, tuplas, outras listas,dicionários, dentre outros. Revisaremos as principais questões que envolvem este tópico.

Introdução

Vamos começar criando uma lista vazia.

In [93]:

```
lista = []
lista
```

Out[93]:

```
[]
```

Ou ainda:

In [94]:

```
lista = list()
lista
```

Out[94]:

```
[]
```

Agora vamos criar uma lista com alguns valores. E em seguida uma lista com tipos distintos de dados.

In [95]:

```
lista1 = [10, 20, 30, 40]
lista1
```

Out[95]:

```
[10, 20, 30, 40]
```

In [96]:

```
lista2 = ['Hello World!', 10, 20, 30, [1, 2, 3], (45, 50)]
lista2
```

Out[96]:

```
['Hello World!', 10, 20, 30, [1, 2, 3], (45, 50)]
```

Observe que a lista2 armazena uma string ('Hello World!'), números inteiros, outra lista e uma tupla.

Para acessar valores em uma lista devemos utilizar o índice do valor a ser acessado. Lembrando que o índice de uma lista começa em 0, isto é, o índice do primeiro elemento.

In [97]:

```
lista3 = ['A', 'B', 'C', 'D']
lista3
```

Out[97]:

```
['A', 'B', 'C', 'D']
```

A string 'A' possui índice 0, já a string 'B' índice 1 e assim sucessivamente até a string 'D', que possui índice 3.

Podemos verificar o tipo de dado da nossa lista3 com a função built-in type().

In [98]:

```
type(lista3)
```

Out[98]:

```
list
```

Acessando elementos de uma lista via índice

Vejamos mais exemplos de como acessar elementos em uma lista.

Acessando a string 'A':

In [99]:

```
lista = ['A', 'B', 'C']
lista[0]
```

Out[99]:

```
'A'
```

Acessando a string 'B':

In [100]:

```
lista[1]
```

Out[100]:

```
'B'
```

Acessando a string 'C':

In [101]:

```
lista[2]
```

Out[101]:

```
'C'
```

Se executarmos lista[3] teremos IndexError: list index out of range, pois não temos nenhum valor ocupando este índice.

Modificação dos elementos de uma lista

Vamos mudar todos os valores da lista a seguir.

In [102]:

```
a = [10, 11, 12, 13]
```

In [103]:

```
a[0] = 0  #mudando o elemento de índice 0, ou seja, o 10
```

In [104]:

```
a[1] = 0  #mudando o elemento de índice 1, ou seja, o 11
```

In [105]:

```
a[2] = 0  #mudando o elemento de índice 2, ou seja, o 12
```

In [106]:

```
a[3] = 0  #mudando o elemento de índice 3, ou seja, o 13
```

Vejamos o resultado:

In [107]:

```
a
```

Out[107]:

```
[0, 0, 0, 0]
```

Outra forma de indexação

O último elemento de uma lista possui índice -1.

In [108]:

```
b = ['a', 'c', 'd', 'h']
b
```

Out[108]:

```
['a', 'c', 'd', 'h']
```

Vamos pegar o último elemento da lista b:

In [109]:

```
b[-1]
```

Out[109]:

```
'h'
```

Então, podemos acessar os elementos de uma lista de forma contrária (indexação negativa), do fim para o início. Na nossa lista b teríamos o seguinte:

- 'h' - índice **-1**
- 'd' - índice **-2**
- 'c' - índice **-3**
- 'a' - índice **-4**

Na prática:

In [110]:

```
b[-2]
```

Out[110]:

```
'd'
```

In [111]:

```
b[-3]
```

Out[111]:

```
'c'
```

In [112]:

```
b[-4]
```

Out[112]:

```
'a'
```

Tamanho de uma lista

Para verificar o tamanho de uma lista usamos a função integrada len().

In [113]:

```
z = ['a', 'b', 'k', 'j']
len(z)
```

Out[113]:

4

Máximo, mínimo, soma e tamanho de uma lista

Vamos verificar os valores máximo e mínimo da lista a seguir, além da soma de todos os seus elementos e o seu tamanho.

In [114]:

```
num = [31, 914, 236, 376, 140, 705 ]
```

In [115]:

```
print(f'Valor máximo: {max(num)}')
print(f'Valor mínimo: {min(num)}')
print(f'Números de elementos: {len(num)}')
print(f'Soma dos elementos: {sum(num)}')
```

```
Valor máximo: 914
Valor mínimo: 31
Números de elementos: 6
Soma dos elementos: 2402
```

Como unir listas

Podemos unir duas listas com o operador +, como segue:

In [116]:

```
lista1 = ['a', 'b', 'c']
lista2 = ['d', 'e', 'f', 'g']
uniao = lista1+lista2
print(uniao)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Repetindo elementos

Podemos repetir os elementos de uma lista usando o operador *:

In [117]:

```
h = [0,1,2]
h*4
```

Out[117]:

```
[0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2]
```

Construtor list()

Podemos construir listas com o construtor **list()**.Exemplos:

In [118]:

```
tupla = (1, 2, 3, 4, 5)
lista = list(tupla)
```

Ou seja, passamos uma tupla dentro do construtor **list()**, que será convertida para o tipo **list**.

In [119]:

```
lista
```

Out[119]:

```
[1, 2, 3, 4, 5]
```

In [120]:

```
type(lista)
```

Out[120]:

```
list
```

Podemos criar uma lista com uma sequência de números inteiros por meio da função embutida range().

- Sintaxe: range(início, fim, passo)

In [121]:

```
serie1 = list(range(2010, 2021))
serie1
```

Out[121]:

```
[2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]
```

Passamos o range() dentro do construtor list(), de modo que teremos uma sequência de números inteiros de 2010 a 2021 (exclusive).

Outros exemplos:

In [122]:

```
serie2 = list(range(11))
serie2
```

Out[122]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [123]:

```
serie3 = list(range(3,11))
serie3
```

Out[123]:

```
[3, 4, 5, 6, 7, 8, 9, 10]
```

In [124]:

```
serie4 = list(range(1950,2021,10))
serie4
```

Out[124]:

```
[1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]
```

Convertendo uma string em uma lista

Podemos usar a função split() dividir uma string em uma lista, sendo que cada palavra desta string será um item da lista. Para tanto usamos um delimitador (argumento sep). Exemplos:

In [125]:

```
frase = 'Eu amo aprender Python!'
frase.split(sep = ' ')
```

Out[125]:

```
['Eu', 'amo', 'aprender', 'Python!']
```

In [126]:

```
y = 'python%python%python%python'
y.split(sep = '%')
```

Out[126]:

```
['python', 'python', 'python', 'python']
```

In [127]:

```
z = 'carro,computador,livro,café'
z.split(',')
```

Out[127]:

```
['carro', 'computador', 'livro', 'café']
```

Transformando uma lista em uma string

Podemos juntar strings dentro de uma lista, usando um caractere como separador, com a função **join()**.

In [128]:

```
lista = ['Eu','amo','python!']
string = ' '.join(lista)

string
```

Out[128]:

```
'Eu amo python!'
```

Definindo listas em termos de variáveis

Vamos declarar as variáveis num1,num2,num3 e armazenar em lista.

In [129]:

```
num1 = 10
num2 = 20
num3 = 30
lista = [num1, num2, num3]

lista
```

Out[129]:

```
[10, 20, 30]
```

Desempacotando listas

Podemos ainda desempacotar os valores de uma lista em variáveis.

In [130]:

```
valores = [350, 400, 900, 120]
num1, num2, num3, num4 = valores
```

In [131]:

```
num1
```

Out[131]:

```
350
```

In [132]:

```
num2
```

Out[132]:

```
400
```

In [133]:

```
num3
```

Out[133]:

```
900
```

In [134]:

```
num4
```

Out[134]:

```
120
```

Este notebook objetiva demonstrar outras operações efetuadas em listas.

Fatiamento de listas (*slicing*)

Considere a lista hipotética z. Se quisermos selecionar todos os valores entre os índices m-n (selecionar todos os valores começando no valor m até o valor de índice n-1):

- z[m:n]

Para ilustrar este conceito vamos declarar a lista *z* e em seguida efetuar algumas operações.

In [135]:

```
z = [4, 7, 9, 11, 3, 1]
z
```

Out[135]:

```
[4, 7, 9, 11, 3, 1]
```

Na prática, vamos pegar os elementos que começam no índice 1 e terminam no índice 4, mas lembre-se que é exclusive, (4-1).

In [136]:

```
z[1:4]
```

Out[136]:

```
[7, 9, 11]
```

Vamos pegar todos os elementos que começam no índice 1 até o final da nossa lista.

In [137]:

```
z[1:]
```

Out[137]:

```
[7, 9, 11, 3, 1]
```

Então, lembre-se que para fatiar uma lista podemos usar essa definição inicial:

- lista[início:fim]

Vamos pegar todos os elementos que estão no início da lista (índice zero) até o índice 3.

In [138]:

```
z[:3]
```

Out[138]:

```
[4, 7, 9]
```

Vamos atualizar nossa definição de fatiamento para:

- lista[início:fim:passo]

Pense no 'passo' (ou step) como sendo um número inteiro que especifica o incremento entre cada índice do nosso slicing. Ou seja, podemos pegar todos os elementos entre um intervalo de 2 em 2, de 5 em 5, e assim por diante.

Então, vamos mostrar todos os valores da nossa lista anterior de 2 em 2.

In [139]:

```
z[::2]
```

Out[139]:

```
[4, 9, 3]
```

Vamos criar uma outra lista e mostrar outras possibilidades.

In [140]:

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

Out[140]:

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

Vamos pegar todos os elementos da nossa lista 'p' que começam no índice 1 até o 7, de 2 em 2.

In [141]:

```
p[1:7:2]
```

Out[141]:

```
[2, 4, 6]
```

Exemplos com índice negativo:

In [142]:

```
p[-10:-5]
```

Out[142]:

```
[1, 2, 3, 4, 5]
```

Ou seja, pegamos todos os elementos do índice -5 até o índice -10 (ou p[0:5]).

Que é o mesmo que:

In [143]:

```
p[:5]
```

Out[143]:

```
[1, 2, 3, 4, 5]
```

Agora vamos pegar os valores entre os índices -10 a -5 de dois em dois.

In [144]:

```
p[-10:-5:2]
```

Out[144]:

```
[1, 3, 5]
```

Podemos usar esse conceito para inverter nossa lista:

In [145]:

```
p[::-1]
```

Out[145]:

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

Vamos inverter e pegar os elementos de 2 em 2 e depois de 3 em 3:

In [146]:

```
p[::-2]
```

Out[146]:

```
[10, 8, 6, 4, 2]
```

In [147]:

```
p[::-3]
```

Out[147]:

```
[10, 7, 4, 1]
```

Iterando em um lista

Podemos percorrer todos os elementos de uma lista com o loop *for* e efetuar diversas operações. Vejamos um exemplo em que percorremos todos os elementos de uma lista e mostramos na tela cada um de seus elementos.

In [148]:

```
lista_carros = ['Uno', 'Gol', 'Argo', 'Celta', 'Palio', 'Mobi', 'Up']
```

In [149]:

```
for carro in lista_carros:
    print(carro)
```

```
Uno
Gol
Argo
Celta
Palio
Mobi
Up
```

Vamos mostrar o quadrado de cada número da lista *lista_num* na tela.

In [150]:

```
lista_num = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for numero in lista_num:
    print(numero**2)
```

```
1
4
9
16
25
36
49
64
81
100
```

A partir dos números da lista a seguir, vamos classificar os números em pares e ímpares e mostrar o resultado na tela.

In [151]:

```
numeros = [1, 2, 3, 4, 5]
for numero in numeros:
    if numero%2==0:
        print(numero, 'Par')
    else:
        print(numero,'Ímpar')
```

```
1 Ímpar
2 Par
3 Ímpar
4 Par
5 Ímpar
```

Vamos somar todos os números da *lista1*. Para tanto, vamos criar uma variável soma inicializada com valor nulo, para acumular os valores a cada laço de repetição.

In [152]:

```
lista1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
soma = 0
for num in lista1:
    soma+=num  #equivale a soma = soma+num
print(soma)
```

```
55
```

Utilizando a função sum do Python teremos o mesmo resultado:

In [153]:

```
print(sum(lista1))
```

```
55
```

Vamos calcular a média dos números da lista 'numeros'.

Uma forma mais simples seria:

In [154]:

```
numeros = [10, 8, 7, 9, 7]
mean = sum(numeros)/len(numeros)  #cálculo da média
print(f'Média: {mean}')
```

```
Média: 8.2
```

Utilizando loop while:

In [155]:

```
numeros = [10, 8, 7, 9, 7]
soma = 0  #acumulador
x = 0  #contador
while x<len(numeros):  #condição lógica
    soma+=numeros[x]  #acumula cada valor da lista numeros no índice x
    x+=1  #incrementa o contador a cada laço
print(f'Média: {soma/x}')
```

```
Média: 8.2
```

Verificando se um elemento está contido em uma lista

Podemos verificar se um determinado elemento está contido em uma lista com a keyword **in**.

In [156]:

```
k = ['F', 'M', 'J', 'K']  #declara a lista
```

In [157]:

```
'F' in k  #verifica se o elemento 'F' está contido na lista
```

Out[157]:

```
True
```

In [158]:

```
'F' not in k  #verifica se o elemento 'F' não está contido na lista
```

Out[158]:

```
False
```

In [159]:

```
'Z' in k  #verifica se o elemento 'Z' está contido na lista
```

Out[159]:

```
False
```

In [160]:

```
'Z' not in k  #verifica se o elemento 'Z' não está contido na lista
```

Out[160]:

```
True
```

Podemos criar uma estrutura condicional para verificar se um elemento está contido em uma lista e caso seja uma condição verdadeira, mostrar uma determinada mensagem na tela.

In [161]:

```
if 'M' in k:
    print('M pertence a lista k.')
else:
    print('M não pertence a lista k.')
```

```
M pertence a lista k.
```

Nesta parte do notebook veremos os principais métodos de listas.

Métodos de Listas

Métodos em List	Descrição
append()	Adicona um elemento ao final da lista.
insert()	Adiciona um elemento em um índice especificado.
extend()	Adiciona os elementos de um iterável no final da lista.
pop()	Remove e retorna um elemento de um índice especificado.
remove()	Remove o primeiro elemento com o valor especificado.
clear()	Remove todos os elementos de uma lista.
copy()	Retorna uma cópia da lista selecionada.
sort()	Ordena a lista.
reverse()	Reverte a ordem dos elementos em uma lista.
index()	Retorna o primeiro índice do elemento especificado.
count()	Conta o número de vezes que o elemento aparece na lista.

append - adiciona um elemento ao final da lista.

In [162]:

```
numeros =[45, 50, 55]  #declara a lista
numeros
```

Out[162]:

```
[45, 50, 55]
```

In [163]:

```
numeros.append(60)  #adiciona o número 60 ao final da lista
numeros
```

Out[163]:

```
[45, 50, 55, 60]
```

Se fizermos numeros.append(60,61) teremos:

- TypeError: append() takes exactly one argument (2 given)

Ou seja, o método append recebe apenas um argumento. Poderíamos então colocar os números dentro de uma lista ou tupla e passar como argumento da função append.

In [164]:

```
numeros.append([60,65])
```

In [165]:

```
numeros
```

Out[165]:

```
[45, 50, 55, 60, [60, 65]]
```

In [166]:

```
numeros.append((70,75))
```

In [167]:

```
numeros
```

Out[167]:

```
[45, 50, 55, 60, [60, 65], (70, 75)]
```

extend - adiciona os elementos de um iterável (por exemplo, lista,tupla...) ao final de uma lista.

In [168]:

```
numeros = [0, 0, 0, 0]  #declaramos uma lista
numeros.extend([1, 2, 3])  #adicionamos apenas os números do iterável

numeros
```

Out[168]:

```
[0, 0, 0, 0, 1, 2, 3]
```

In [169]:

```
numeros.extend((4,5,6))  #adicionamos apenas os números da tupla

numeros
```

Out[169]:

```
[0, 0, 0, 0, 1, 2, 3, 4, 5, 6]
```

In [170]:

```
numeros.extend({7,8,9,10})  #adicionamos apenas os números do set

numeros
```

Out[170]:

```
[0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 7]
```

In [171]:

```
numeros.extend('abcdef')  #adicionamos as letras da string

numeros
```

Out[171]:

```
[0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 7, 'a', 'b', 'c', 'd',
'e', 'f']
```

In [172]:

```
numeros.extend(['abc'])

numeros
```

Out[172]:

```
[0,
 0,
 0,
 0,
 1,
 2,
 3,
 4,
 5,
 6,
 8,
 9,
 10,
 7,
 'a',
 'b',
 'c',
 'd',
 'e',
 'f',
 'abc']
```

insert - adiciona um elemento em uma posição especificada. Sintaxe:

- lista.insert(index,object) - primeiro passamos o índice e depois o objeto.

In [173]:

```
numeros = [4, 10, 11, 12, 7, 9, 15]  #declara a lista

numeros
```

Out[173]:

```
[4, 10, 11, 12, 7, 9, 15]
```

In [174]:

```
numeros.insert(1,0)  #insere o número 0 no índice 1

numeros
```

Out[174]:

```
[4, 0, 10, 11, 12, 7, 9, 15]
```

In [175]:

```
numeros.insert(0, 11)  #insere o número 11 no índice 0

numeros
```

Out[175]:

```
[11, 4, 0, 10, 11, 12, 7, 9, 15]
```

remove - remove o primeiro item com o valor especificado.

ValueError: se inserirmos um valor que não esteja na lista teremos esse tipo de erro.

In [176]:

```
numeros = [10, 11, 12, 13]
numeros
```

Out[176]:

```
[10, 11, 12, 13]
```

In [177]:

```
numeros.remove(13)  #removemos o número 13 da lista
numeros
```

Out[177]:

```
[10, 11, 12]
```

In [178]:

```
numeros.remove(12)  #removemos o número 12 da lista
numeros
```

Out[178]:

```
[10, 11]
```

In [179]:

```
numeros.remove(10)  #remove o número 10 da lista
numeros
```

Out[179]:

```
[11]
```

pop - remove e retorna o elemento com o índice especificado.

In [180]:

```
numeros = ['k', 'm', 'z', 'a', 'b', 'o']
numeros
```

Out[180]:

```
['k', 'm', 'z', 'a', 'b', 'o']
```

In [181]:

```
numeros.pop()  #se não inserirmos o índice, o último elemento será removido
```

Out[181]:

```
'o'
```

In [182]:

```
numeros
```

Out[182]:

```
['k', 'm', 'z', 'a', 'b']
```

In [183]:

```
numeros.pop(1)  #remove e retorna o elemento com índice 1
```

Out[183]:

```
'm'
```

In [184]:

```
numeros
```

Out[184]:

```
['k', 'z', 'a', 'b']
```

In [185]:

```
numeros.pop(3)
```

Out[185]:

```
'b'
```

In [186]:

```
numeros
```

Out[186]:

```
['k', 'z', 'a']
```

Também podemos deletar elementos de uma lista com a instrução del:

In [187]:

```
letras = ['k', 'i', 'j', 'm']
del letras[0]  #deletamos o item de índice 0
```

In [188]:

```
letras
```

Out[188]:

```
['i', 'j', 'm']
```

Podemos deletar completamente a lista:

In [189]:

```
del letras
```

Já não temos mais a lista *letras*. Se tentarmos acessá-la novamente teremos: NameError.

clear - remove todos os elementos da lista.

In [190]:

```
letras = ['abc','ab','a']
letras
```

Out[190]:

```
['abc', 'ab', 'a']
```

In [191]:

```
letras.clear() # aplicando método

letras
```

Out[191]:

```
[]
```

count - retorna o número de ocorrências de um valor especificado.

In [192]:

```
numeros = [1, 1, 1, 0, 0, 0, 5, 3, 1, 2]
```

In [193]:

```
numeros.count(1)  #conta o número de ocorrências do número 1
```

Out[193]:

```
4
```

In [194]:

```
numeros.count(0)  #conta o número de ocorrências do número 0
```

Out[194]:

```
3
```

In [195]:

```
numeros.count(5)  #conta o número de ocorrências do número 5
```

Out[195]:

```
1
```

In [196]:

```
numeros.count(10)  #conta o número de ocorrências do número 10
```

Out[196]:

```
0
```

reverse - inverte a ordem da lista.

In [197]:

```
numeros = [10, 11, 12, 13, 14, 15]

numeros
```

Out[197]:

```
[10, 11, 12, 13, 14, 15]
```

In [198]:

```
numeros.reverse()  #inverte a ordem da listan
numeros
```

Out[198]:

```
[15, 14, 13, 12, 11, 10]
```

In [199]:

```
letras = ['a','b','c','d']

letras
```

Out[199]:

```
['a', 'b', 'c', 'd']
```

In [200]:

```
letras.reverse()
letras
```

Out[200]:

```
['d', 'c', 'b', 'a']
```

In [201]:

```
letras.reverse()  #inverte a ordem da lista
letras
```

Out[201]:

```
['a', 'b', 'c', 'd']
```

sort - ordena a lista.

In [202]:

```
numeros = [-10, 100, -1, 60, 76, 33, -30, 150]  #declara uma lista não ordenada

numeros
```

Out[202]:

```
[-10, 100, -1, 60, 76, 33, -30, 150]
```

In [203]:

```
numeros.sort()  #ordena a lista

numeros
```

Out[203]:

```
[-30, -10, -1, 33, 60, 76, 100, 150]
```

In [204]:

```
anos = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]  #declara uma lista não
 ordenada

anos.sort(reverse=True)
```

In [205]:

```
anos
```

Out[205]:

```
[2020, 2010, 2000, 1990, 1980, 1970, 1960, 1950]
```

index - retorna o índice do primeiro elemento com o valor especificado. Sintaxe:

- index(value, start=0,stop)

In [206]:

```
lista = [5, 6, 7, 8]
```

In [207]:

```
lista.index(5)  #retorna o índice da primeira ocorrência do número 5
```

Out[207]:

```
0
```

In [208]:

```
lista.index(7)  #retorna o índice da primeira ocorrência do número 7
```

Out[208]:

```
2
```

In [209]:

```
numeros = [1, 2, 3, 3, 3, 3, 3, 7, 7, 8, 9, 10, 10, 11, 12, 11]
numeros.index(3)  #retorna o índice da primeira ocorrência do número 3
```

Out[209]:

```
2
```

In [210]:

```
numeros.index(3, 5)  #retorna o índice do valor 3, começando a contar a partir d
o índice 5 da lista
```

Out[210]:

```
5
```

copy - retorna uma cópia da lista. Existem duas formas de cópia em Python: Shallow Copy e Deep Copy.

Shallow Copy

In [211]:

```
lista_numeros = [4, 9, 11, -1]
nova = lista_numeros  #cria uma cópia de lista_numeros

nova
```

Out[211]:

```
[4, 9, 11, -1]
```

In [212]:

```
nova.append(0)  #acrescenta um número na lista nova
```

In [213]:

```
nova
```

Out[213]:

```
[4, 9, 11, -1, 0]
```

Este número também será acrescentado em *lista_numeros*.

In [214]:

```
lista_numeros
```

Out[214]:

```
[4, 9, 11, -1, 0]
```

Ou seja, adicionamos o número 0 na cópia (lista *nova*), como também na lista original. Se acrescentarmos um valor na nossa lista original, teremos o mesmo comportamento.

In [215]:

```
lista_numeros.append(100)  #acrescenta o número 100
```

O resulta na lista *nova* será:

In [216]:

```
lista_numeros
```

Out[216]:

```
[4, 9, 11, -1, 0, 100]
```

Deep Copy (cópia profunda)

In [217]:

```
lista_numeros = [1, 2, 3]
nova = lista_numeros.copy()  #deep copy
```

In [218]:

```
nova
```

Out[218]:

```
[1, 2, 3]
```

In [219]:

```
nova.append(4)  #adiciona o número 4 na cópia
```

In [220]:

```
nova
```

Out[220]:

```
[1, 2, 3, 4]
```

In [221]:

```
lista_numeros  #a lista original não é afetada
```

Out[221]:

```
[1, 2, 3]
```

Assim, acrescentamos o número 4 na nossa cópia, mas esse valor não foi adicionado na lista original.

Tuplas

Tuplas são coleções ordenadas e imutáveis. Existem dois métodos para tuplas: count e index. Podemos criar tuplas com ():

In [222]:

```
tupla = (1, 2, 3, 4, 5)
tupla
```

Out[222]:

```
(1, 2, 3, 4, 5)
```

Outras formas de criar tuplas:

In [223]:

```
tupla1 = 1, 2, 3, 4, 5
tupla1
```

Out[223]:

```
(1, 2, 3, 4, 5)
```

In [224]:

```
tupla2 = tuple([1, 2, 3])  #passamos uma lista dentro do construtor tuple()
tupla2
```

Out[224]:

```
(1, 2, 3)
```

In [225]:

```
tupla3 = tuple({1, 2, 3})  #passamos um set dentro do construtor tuple
tupla3
```

Out[225]:

```
(1, 2, 3)
```

In [226]:

```
tupla4 = tuple(range(1, 11))  #passamos um range() dentro do construtor tuple
tupla4
```

Out[226]:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

A título de curiosidade, veja o tipo do objeto "tupla" quando é criado.

In [227]:

```
type(tupla)  # checa o tipo do objeto criado
```

Out[227]:

```
tuple
```

Forma incorreta de declarar uma tupla com um elemento:

In [228]:

```
tupla = (4)
tupla
```

Out[228]:

```
4
```

Pode parecer que a tupla foi criada, mas quando você analisar o "tipo" do objeto criado, ele não será uma tupla.

In [229]:

```
type(tupla)
```

Out[229]:

```
int
```

Quando se precisa criar uma tupla com apenas um valor, a forma correta são as seguintes.

In [230]:

```
tupla = (4,)
tupla
```

Out[230]:

```
(4,)
```

Ou ainda:

In [231]:

```
tupla = 4,
tupla
```

Out[231]:

```
(4,)
```

Se verificar o tipo do objeto, verá que foram criados tuplas.

In [232]:

```
type(tupla)
```

Out[232]:

```
tuple
```

Desempacotamento de Tupla

Podemos desempacotar os valores de uma tupla em variáveis.

In [233]:

```
valores = (40, 100, 200)
num1, num2, num3 = valores  #desempacota a tupla em 3 variáveis
```

In [234]:

```
num1
```

Out[234]:

```
40
```

In [235]:

```
num2
```

Out[235]:

```
100
```

In [236]:

```
num3
```

Out[236]:

```
200
```

Valor máximo e mínimo, soma e tamanho de uma tupla

Usamos as funções built-in min(),max(),sum() e len() para obter, respectivamente: valor mínimo, valor máximo, soma dos elementos e tamanho de uma tupla.

In [237]:

```
numeros = (10, -37, 41, 88, 91, 13, 9, 3, 62)
```

In [238]:

```
max(numeros)  #valor máximo da tupla
```

Out[238]:

```
91
```

In [239]:

```
min(numeros)  #valor mínimo da tupla
```

Out[239]:

```
-37
```

In [240]:

```
sum(numeros)  #soma dos valores da tupla
```

Out[240]:

```
280
```

In [241]:

```
len(numeros)  #tamanho da tupla
```

Out[241]:

```
9
```

Unindo tuplas

Podemos usar o operador **+** para unir das tuplas.

In [242]:

```
tupla1 = ('a', 'b', 'h')
tupla2 = (40, 3, 11)
tupla3 = tupla1+tupla2
```

In [243]:

```
tupla3
```

Out[243]:

```
('a', 'b', 'h', 40, 3, 11)
```

Iterando em tuplas

Podemos usar um loop for para percorrer todos os elementos de uma tupla. Abaixo percorremos os números da tupla e mostramos na tela seus respectivos valores.

In [244]:

```
tupla = (1, 2, 3, 4, 5, 6)
for numero in tupla:
    print(numero)
```

```
1
2
3
4
5
6
```

Método em Tuplas

Temos basicamente dois métodos aplicáveis as tuplas:

Método	Descrição
index()	Retorna o índice do elemento especificado.
count()	Conta o número de vezes que o elemento aparece na tupla.

Usando o método index:

In [245]:

```
tupla = ('a', 'b', 'c', 'd')
print(f"Índice de 'a': {tupla.index('a')}")
print(f"Índice de 'b': {tupla.index('b')}")
print(f"Índice de 'c': {tupla.index('c')}")
print(f"Índice de 'd': {tupla.index('d')}")
```

```
Índice de 'a': 0
Índice de 'b': 1
Índice de 'c': 2
Índice de 'd': 3
```

Ou por meio de um loop for:

In [246]:

```
for letra in tupla:
    print(f'Índice de {letra}: {tupla.index(letra)}')
```

```
Índice de a: 0
Índice de b: 1
Índice de c: 2
Índice de d: 3
```

Usando o método count:

In [247]:

```
tupla = ('a', 'z', 'z', 'z', 'w')
print(tupla.count('z'))
```

```
3
```

More Academy