

Comprehensions: permitem uma notação condensada para manipulação de listas, dicionários e conjuntos, eliminando sem a necessidade de loops e condicionais externos, resultando em um tempo de processamento computacional mais eficiente (MENEZES, 2020).

Set Comprehensions: é um mecanismo conciso para criar conjuntos de elementos que atendam a uma determinada condição. É similar às *list comprehensions*, mas, em vez de criar listas, elas criam conjuntos. Os conjuntos, diferentemente das listas, não mantêm uma ordenação de seus elementos e não permitem duplicatas.

Exemplo: sintaxe básicaExemplo:

```
{expressao for item in iteravel if condicao}
```

- **expressao** é o valor que será incluído no conjunto resultante.
- **item** é a variável que representa o elemento atual durante a iteração.
- **iteravel** é um objeto sobre o qual a iteração é realizada (ex: lista, tupla, string, etc.).
- **condicao** é uma expressão booleana opcional que determina se o item deve ser incluído no conjunto. Fonte: Referência bibliográfica não fictícia: **Python 3 documentation. "5. Data Structures"**. Python Software Foundation.

Url: <https://docs.python.org/3/tutorial/datastructures.html#sets>

Vantagens Set Comprehension:

- **Legibilidade e Concisão:** permite a criação de conjuntos complexos de uma maneira concisa e legível em comparação com o uso de loops e condicionais explícitos.
- **Performance:** são geralmente mais rápidas do que a criação manual de conjuntos usando loops for, especialmente para conjuntos maiores, porque a operação é otimizada e executada internamente em C, mesmo quando estamos utilizando a linguagem Python.
- **Eliminação de Duplicatas:** permite criar uma coleção de dados garantindo que cada elemento seja único.
- **Expressividade:** permite a incorporação de condições e transformações, além facilitar a filtragem itens que não deseja e transformar itens conforme necessário em uma única linha de código.
- **Versatilidade:** podem ser utilizadas para realizar uma ampla variedade de tarefas, como filtragem de dados, transformações de elementos e operações matemáticas.

Casos de uso:

- **Remoção de Duplicatas de uma Sequência:** ao lidar com dados coletados de fontes externas, você pode encontrar sequências com elementos duplicados. A compreensão de conjuntos pode ser utilizada para remover essas duplicatas de maneira eficiente. Exemplo

```
# Dada a lista
nomes = ['Gertrudes', 'Asdrobaldo', 'Gertrudes', 'Masterlandia', 'Asdrobaldo', 'Genoveva']

# Set comprehension - removendo duplicatas
nomes_unicos = {nome for nome in nomes}

print(f'\nOs nomes sem duplicatas são: {nomes_unicos} \n')
```

```
Os nomes sem duplicatas são: {'Gertrudes', 'Masterlandia', 'Genoveva', 'Asdrobaldo'}
```

- **Transformações e Operações Matemáticas:** podem ser usadas para aplicar uma transformação ou realizar operações matemáticas em todos os elementos de uma coleção.

- Cria um conjunto de seus quadrados:

```
# Dada uma lista
numeros = [1, 2, 3, 4, 5]

quadrados = {n**2 for n in numeros}

print(f'\nO quadrado dos números da lista é: {quadrados} \n')
```

```
O quadrado dos números da lista é: {1, 4, 9, 16, 25}
```

- **Filtragem de Dados:** dados de acordo com determinadas condições, este tipo de tarefa é muito utilizado na limpeza e processamento.

Filtragem de dados:

```
# Dada uma lista de números
numeros = [1, -4, -2, 3, -1, 5, -6]

# filtra apenas números positivos
positivos = {n for n in numeros if n > 0}
print(f'\nOs números positivos são: {positivos} \n')
```

```
Os números positivos são: {1, 3, 5}
```

listas aninhadas e *comprehensions*; expressões lambdas λ e funções Integradas.

- As Listas aninhadas ou ***nested lists*** possibilita a definição de uma matriz multidimensional com dados de diferentes tipos.

Lista aninhada - matriz 4 X 4 e iterando com for() e range()

```
matriz = [[numero for numero in range(1, 5)] for valor in range(1, 5)]

print(f'\nA matriz original é: {matriz} \n')

print(f'ou \n')

[[print(dados) for dados in matriz]]

print()
```

```
A matriz original é: [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]

ou

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

Lista aninhada: matriz 3 x 3 de nomes - combinando os elementos das sublistas de modo que o primeiro elemento de cada sublista seja agrupado com o primeiro elemento segundo elemento de cada sublista:

```
listas = [[1, 2, 3], ['Josefina', 'Astrogildo', 'Gestronilda'], ['F', 'M', 'F']]
print(f'\nA lista aninhada é: {listas}')

# Usando zip em conjunto com list comprehension

agrupados = [' '.join(map(str, tupla)) for tupla in zip(*listas)]

print(f'\nOs elementos agrupados são: {" "; ".join(agrupados)}\n')
```

```
A lista aninhada é: [[1, 2, 3], ['Josefina', 'Astrogildo', 'Gestronilda'], ['F', 'M', 'F']]

Os elementos agrupados são: 1 Josefina F; 2 Astrogildo M; 3 Gestronilda F
```

EXPRESSÕES LAMBDA λ

As expressões lambda λ constituem uma abstração matemática fundamental na programação funcional. Seu uso principal é resolver problemas matemáticos dentro de aplicações que adotam este paradigma.

Originado na década de 1930 por Alonzo Church, o cálculo lambda λ antecede a era dos computadores, debruçando-se sobre os fundamentos teóricos da computação. Church, em colaboração com figuras notáveis como Haskell Curry, Kurt Gödel, Emil Post e Alan Turing, estabeleceu uma definição formal de algoritmos (Mueller, 2020, p.132).

O propósito do cálculo lambda λ é explorar a abstração funcional sob uma ótica abstrata e matemática. Tal abstração inicia com a decomposição de problemas em etapas lógicas e sequenciais. Contudo, se uma etapa não pode ser traduzida em função, perde-se sua utilidade, pois a essência de aplicar uma função a um argumento é gerar um valor resultante. Por isso, a compreensão dos princípios do cálculo lambda é vital para entender as linguagens de programação contemporâneas:

- Expressões lambda λ são construídas exclusivamente com funções, dispensando outros tipos de dados como strings, inteiros, booleanos, entre outros, presentes em linguagens atuais. Todos os tipos são representados através de funções, estabelecendo-as como alicerce de todo o sistema.

- Estas expressões são isentas de estado e efeitos colaterais, permitindo sua representação através de um modelo substitutivo, onde a ordem de avaliação se torna insignificante. Embora as linguagens modernas adotem uma ordem específica para simplificar a avaliação de funções, é importante notar que as expressões lambda λ são unárias, requerendo um único argumento. Funções que necessitam de múltiplos argumentos recorrem ao uso de funções *curried* (Mueller, 2020).

Ao desenvolver programa usando expressão Lambda λ , normalmente usamos três operações diferentes:

- 1) Definição de funções como variáveis.
- 2) Atribuição de uma variável à expressão (abstração).
- 3) Aplicar de uma função a um argumento. (MUELLER, 2020 - pg 135).

As variáveis nas expressões lambda λ representam espaços reservados para valores, constituindo termos lambda λ . Elas são a pedra angular da definição indutiva dos termos lambda, apoiando-se no princípio de que se uma propriedade é válida para um determinado número natural (n), será também para o próximo ($n + 1$).

Além disso, nas expressões lambda λ , as variáveis podem ou não ser tipadas. Neste contexto, a tipagem não denota um tipo de dado específico, mas sim delinea a interpretação do cálculo lambda λ (Mueller, 2020).

Flexibilidade e precisão das variáveis nas expressões lambda λ em Python

Segundo Mueller (2020), a “flexibilidade” da lambda λ em Python decorre de sua tipagem dinâmica, onde o tipo de uma variável pode mudar durante a execução do programa e isso não está diretamente relacionado à ideia de cálculos não termináveis ou resultados indefinidos. Em Python, tais problemas estão associados a lógicas de programação defeituosas (como loops infinitos mal construídos) do que ao sistema de tipos da linguagem.

As expressões lambda λ proporcionam uma estrutura bem definida. Considerando dois termos lambda, λM e N , onde M atua como função e N como entrada, a aplicação de N em M é representada por $(M)N$. Os parênteses desempenham um papel crucial, funcionando como operadores de aplicação que direcionam a ordem e a maneira como os termos são aplicados.

```
# Função comum
def funcao_1(x):
    return x + 1

print(f'\n0 valor retornado é: {funcao_1(9)} \n')
```

```
# Função lambda com variáveis tipadas

funcao_1 = lambda x : x + 1

print(f'\n0 valor retornado é: {funcao_1(9)} \n')
```

0 valor retornado é: 10

Dentro deste sistema, onde tudo é função, uma expressão como $M_2(M_1N)$ exemplifica a aplicação de M_1 em N e, subsequentemente, M_2 em M_1 . É importante notar que, embora seja possível encontrar expressões lambda escritas sem parênteses, como em EFG , há uma regra de associação padrão que interpreta esta expressão como $((E)F)G$, indicando que E é aplicado em F e o resultado é aplicado em G . A utilização de parênteses é, portanto, uma prática recomendada para evitar ambiguidades (Mueller, 2020).

Expressões lambda λ com mais de um parâmetro de entrada

```
nome_completo = lambda nome, sobrenome : nome.strip().title() + " " + sobrenome.strip()

print(f'\nInstituição: {nome_completo("IFRO", "Campus Ariquemes")}')
```

```
print(f'\nCurso: {nome_completo(" Análise", "e Desenvolvimento de Sistemas – ADS")} \n')
```

Instituição: Ifro Campus Ariquemes

Curso: Análise e Desenvolvimento de Sistemas – ADS

Ordenando listas usando expressão lambda λ

```
atores = ["Leonardo DiCaprio",
          "Brad Pitt",
          "Dezel Washington",
          "Philip Seymour Hoffman",
          "Will Smith",
          "Johnny Depp"]
atores.sort(key = lambda sobrenome: sobrenome.split(" ")[-1].lower())

print(f'\nLista ordenada: {atores} \n')
```

Lista ordenada: ['Johnny Depp', 'Leonardo DiCaprio', 'Philip Seymour Hoffman', 'Brad Pitt', 'Will Smith', 'Dezel Washington']

Funções quadráticas normalmente são aplicadas no desenvolvimento de jogos quando é necessário calcular a trajetória de naves, foguetes e tiros por exemplo. Em matemática, a função quadrática, também é chamada de função do segundo grau e é expressa: $f(x) = ax^2 + b \cdot x + c$, sendo que os coeficientes "a, b e c" números reais e "a" diferente de 0 (zero).

O grau da função é determinado de acordo com o maior expoente da variável x. No caso da função quadrática, dois é o maior expoente de x. Em **Python** a notação da função quadrática é: **lambda x: a * x ** 2 + b * x + c**

lambda λ para calcular funções quadráticas:

```
def funcao_quadratica(a, b, c):
    return lambda x : a * x ** 2 + b * x + c

quadratica = funcao_quadratica(2, 3, -5)

print(f'\nFunção quadrática de 0 = {quadratica(3)}')
```

```
print(f'\nFunção quadrática de 1 = {quadratica(6)}')
```

```
print(f'\nFunção quadrática de 2 = {quadratica(9)} \n')
```

Função quadrática de 0 = 22

Função quadrática de 1 = 85

Função quadrática de 2 = 184

Lambda λ em Python normalmente é aplicada na filtragem e ordenação de dados, usando técnicas da programação funcional. Também costumamos utilizar lambda λ dentro de outras funções, para evitar escrever duas funções.

Lambda λ para evitar escrever duas funções

```
def somar(x):  
    funcao_2 = lambda x : x + 9  
    return funcao_2(x) * 1  
  
print(f'\n0 valor retornado é: {somar(4)}')
```

0 valor retornado é: 13

O principal benefício da lambda λ é que, com elas, podemos desenvolver funções sem precisar usar a palavra reservada 'def'. Com isso as funções são criadas em tempo de execução do código. *Lembre-se, o corpo de uma expressão lambda λ é uma expressão única, não um bloco de instruções.*

Lambda λ - função sem a palavra reservada def()

```
a = lambda x : x * 2  
  
print(f'\n0 Valor da expressão é: {a(6)} \n')
```

0 Valor da expressão é: 12

Diferentes entradas de dados para funções lambda λ

```
sem_entrada = lambda: 'Estrutura de Dados'  
uma_entrada = lambda x: 5 * x + 3  
duas_entradas = lambda x, y: (x * y) ** 0.8  
tres_entradas = lambda x, y, z: 5 / (2 / x + 2 / y + 2 / z)  
  
print(f'\nLambda sem entrada: {sem_entrada()}')  
print(f'\nLambda com uma entrada: {uma_entrada(12)}')  
print(f'\nLambda com duas entradas: {duas_entradas(7, 9)}')  
print(f'\nLambda com três entradas: {tres_entradas(4, 12, 18)} \n')
```

Lambda sem entrada: Estrutura de Dados

Lambda com uma entrada: 63

Lambda com duas entradas: 27.508850275948053

Lambda com três entradas: 6.42857142857143

Casos de uso expressões lambdas λ

- **Manipulação de Dados com Pandas:** em análise de dados, quando se usa a biblioteca Pandas, as expressões lambda são frequentemente usadas para aplicar uma função a uma coluna inteira de um DataFrame.

- **Processamento de Eventos em Interfaces Gráficas:** em uma aplicação com interface gráfica usando Tkinter, por exemplo, podemos querer vincular um evento de clique de botão a uma função que atualiza algum texto na janela. Uma expressão lambda pode ser usada para passar argumentos adicionais:

```
import tkinter as tk

def atualiza_label(label, texto):
    label.config(text=texto)

root = tk.Tk()
label = tk.Label(root, text="Texto inicial")
label.pack()

# Atualiza o texto do label para "Texto após clique" quando o botão é pressionado
botao = tk.Button(root, text="Clique aqui", command=lambda: atualiza_label(label, "Texto após clique"))
botao.pack()

root.mainloop()
```

Funções Integradas: Na área de desenvolvimento as atividades de transformações estão relacionadas à limpeza e organização dos dados. Neste cenário com frequência precisamos executar ações repetidas vezes, que necessitam da aplicação de uma função específica. Para isso utilizamos as funções *integradas*, pois, elas servem para agilizar os loops de processamentos no Python.

É importante entender que o uso do **for()** deixa o código lento quando o volume de dados é grande. Para dirimir esse problema aplicamos a função **map()** sem precisar utilizar **for()**. A função **map()** devolve os argumentos alterados pela formula da função escolhida no formato que lhe foi repassado.

A sintaxe da função **map()** é: **map(func, seq)**

Para trabalhar com maps precisamos **primeiro criar funções** que serão aplicadas ao tipos de argumentos a partir da função **map()**.

função que calcula a área de um círculo com raio r, com **map()**.

```
import math

def area(r):
    return math.pi * (r ** 2)

raios = [4, 10, 14.2, 0.6, 20, 88]

areas = map(area, raios)

print(f'\nÁreas mapeadas: {areas}')
print(f'\nTipo dos dados mapeados: {type(areas)} \n')
print(f'\nÁrea do círculo calculada: {list(areas)} \n')

print(f'\nO valor das áreas calculadas após a primeira execução de map() é {list(areas)} \n')
```

```
Áreas mapeadas: <map object at 0x10a2f05e0>
Tipo dos dados mapeados: <class 'map'>

Área do círculo calculada: [50.26548245743669, 314.1592653589793, 633.4707426698459, 1.1309733552923256, 1256.6370614359173, 24328.49350939936]

O valor das áreas calculadas após a primeira execução de map() é []
```

Atenção: após a primeira utilização da função **map()** **sozinha**, o container (lista, tupla, conjunto) que recebeu os valores zera (**esvazia**). O Python faz isso para limpar a memória.

Aplicando a função **map()**

- A partir dos iteráveis: dados = **var_1, var_2... var_n**
- Precisaremos de uma função **f(x)**, onde a função **map(f(dados))** irá mapear cada argumento (elemento) de dados e aplicar a função.
- O map object gerado é igual a **f(var_1), f(var_2), f(var_n)**. (F.V.C. 2020, 177)

Função **list()**, **map()** e **lambda()** λ :

```
import math

raios = [4, 10, 14.2, 0.6, 20, 88]

print(f'\nÁreas do círculo calculada {list(map(lambda r : math.pi * (r ** 2), raios))} \n')
```

```
Áreas do círculo calculada [50.26548245743669, 314.1592653589793, 633.4707426698459, 1.1309733552923256, 1256.6370614359173, 24328.49350939936]
```

`List()`, `map()` e `lambda()` λ , para calcular grau Celsius para Fahrenheit

```
temp_celsius = [('Porto Velho', 38), ('Ariquemes', 36), ('Cacoal', 25), ('Vilhena', 20)]

print('\nTemperatura das cidades em graus Celsius: {temp_celsius}')

# f = 9/5 * c + 32

celsius_para_fahrenheit = lambda dado: (dado[0], 9/5 * dado[1] + 32)

print(f'\nTemperatura das cidades em graus Fahrenheit: {list(map(celsius_para_fahrenheit, temp_celsius))} \n')

Temperatura das cidades em graus Celsius: {temp_celsius}

Temperatura das cidades em graus Fahrenheit: [('Porto Velho', 100.4), ('Ariquemes', 96.8), ('Cacoal', 77.0), ('Vilhena', 68.0)]
```

Fontes: (Menezes, 2019; Mueller 2020; F.V.C. 2020).

Filtragem de dados: é uma tarefa essencial para selecionar dados específicos de um container com base em um ou mais critérios. Para utilizar esse recurso aplicamos a função **`filter()`**.

A função `filter()`: recebe dois **parâmetros**, uma função e uma sequência de dados: **`filter(funcao, sequência)`**.

Com ela podemos desenvolver **filtros** usando os **argumentos** em uma sequência de valores de forma otimizada. Essa função retorna **True** sempre que o argumento filtrado for encontrado, (F.V.C. 2020).

Filtrando números pares – `list()`, `filter()` e `lambda` λ

```
numeros = [4, 3, 8, 9, 12, 15, 16, 21]

print(f'\nOs números pares são: {list(filter(lambda nr: nr % 2 == 0, numeros))} \n')
```

Os números pares são: [4, 8, 12, 16]

Selecionando valores acima da média com a função `filter()` e `lambda`

```
import statistics
sensor = [1.3, 2.7, 0.8, 4.1, 4.3, -0.1]

media_sensor = statistics.mean(sensor)

print(f'\nMedia da temperatura capturada pelo sensor: {media_sensor}')

resultado = filter(lambda x: x > media_sensor, sensor)

print('\n0 valores capturados pelo sensor acima da média: {list(resultado)} \n')
```

Media da temperatura capturada pelo sensor: 2.1833333333333333

0 valores capturados pelo sensor acima da média: {list(resultado)}

A diferença entre **map()** e **filter()**:

- A função **map()** é projetada para aplicar uma função especificada a cada item de um iterável, como uma lista, e devolver um novo iterador com os resultados das aplicações dessa função.

- Enquanto, a **filter()** também opera sobre um iterável, utilizando uma função definida para testar a verdade de cada elemento. Contudo, ao invés de transformar itens, ela retorna um novo iterador contendo apenas aqueles elementos para os quais a função de teste retorna verdadeiro.

- Simplificando: **map()** é utilizada para transformar dados, **filter()** é empregada para selecionar dados conforme um critério de filtragem.

Caso de uso:

map() em cálculo de retorno de investimentos: num cenário onde um analista financeiro e tem uma lista de preços de fechamento semanais de uma ação e ele precisa calcular o retorno percentual semanal dessa ação para uma análise subsequente.

O retorno percentual pode ser calculado com a fórmula: $\left(\frac{\text{Preço Atual} - \text{Preço Anterior}}{\text{Preço Anterior}} \right) \times 100$

```
# Lista de preços de fechamento semanais de uma ação
precos_fechamento = [120, 125, 130, 128, 135]

# Função para calcular o retorno percentual
def calcular_retorno(precos):
    preco_anterior, preco_atual = precos
    return (preco_atual - preco_anterior) / preco_anterior * 100

# Usando map() para calcular o retorno percentual
# Zip para criar pares do preço anterior e atual, exceto para o primeiro preço
retornos_percentuais = list(map(calcular_retorno, zip(precos_fechamento, precos_fechamento[1:])))

print(f'\n0 Retorno percentual semanal é: {retornos_percentuais} \n')
```

```
0 Retorno percentual semanal é: [4.166666666666667, 4.0, -1.5384615384615385, 5.46875]
```

- `Filter()` em Tkinter: ao utilizar uma interface gráfica Tkinter e precisar exibir apenas itens de uma lista que atendam a um certo critério, exemplo, todos os itens que contêm a letra 'a', você pode usar `filter()`. Abaixo está um exemplo de como você pode usar `filter()` para atualizar uma lista em um widget `Listbox`:

```
import tkinter as tk

# lista de itens
itens = ['Maçã', 'Banana', 'Cereja', 'Data', 'Fig']

def atualiza_lista_com_filtro():
    letra_a = filter(lambda item: 'a' in item.lower(), itens)
    lista_itens.delete(0, tk.END) # limpa a lista atual
    for item in letra_a:
        lista_itens.insert(tk.END, item) # insere apenas itens com 'a'

# cria janela principal
root = tk.Tk()
root.geometry("300x200") # define o tamanho da janela

# Cria widget listbox
lista_itens = tk.Listbox(root)
lista_itens.place(x=50, y=20, width=200, height=100)

# Botão para filtrar itens
botao_filtro = tk.Button(root, text="Mostrar itens com 'a'", command=atualiza_lista_com_filtro)
botao_filtro.place(x=50, y=130, width=200, height=30)

# Preenche a lista com todos os itens
for item in itens:
    lista_itens.insert(tk.END, item)

root.mainloop()
```

- O `tk.END` é uma constante no módulo `tkinter`. Quando trabalhamos com widgets em `tkinter` que permitem a inserção de texto, como `Listbox`, muitas vezes precisamos inserir um item no final da lista ou do texto já presente. É aqui que o `tk.END` entra em jogo. Ele é uma constante que basicamente diz ao `tkinter` para inserir o novo item ou texto no final do widget. No nosso exemplo: um `Listbox` é usado para mostrar uma lista de itens. Se você quiser adicionar um novo item ao final da lista bastaria usar a sintaxe apresentada.

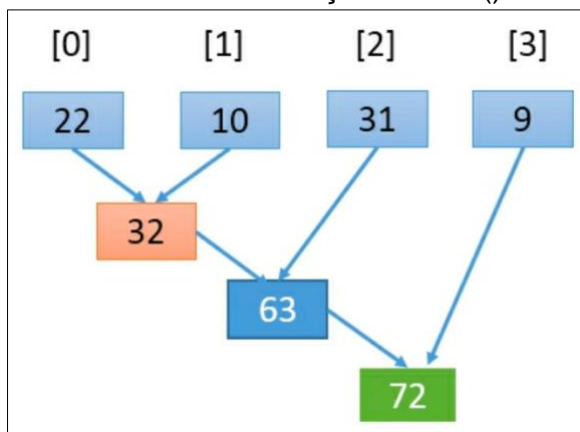
Função `Reduce()`: A função `reduce()` foi modificada a partir da versão 3 do Python. Anteriormente integrada por padrão, agora requer que seja importada da biblioteca `functools` para ser utilizada. Essa função é projetada para receber dois parâmetros: uma função de redução e um iterável. A sintaxe `reduce(fun, seq)` define que o primeiro parâmetro, `fun`, é a função que será aplicada, enquanto o segundo parâmetro, `seq`, é a

sequência sobre a qual a função é aplicada. A `reduce()` opera aplicando a função de redução cumulativamente aos itens da sequência, resultando em um único valor final.

Para detalhes adicionais sobre as funções integradas (built-in functions) do Python, a documentação oficial pode ser consultada no seguinte endereço eletrônico:

<https://docs.python.org/3/library/functions.html>

- Funcionamento da função `reduce()`



Fonte: F. V. C., 2020

Aplicando a função `reduce()`:

```

from functools import reduce

lista = [1, 2, -3, 4, 5, -9]

print(f'\nLista original: {lista} \n')

print('>>> Aplicando a função reduce ()')

def soma(a, b):
    resultado = a + b
    print(f'    Somando {a} + {b} = {resultado}')
    return resultado

resultado_final = reduce(soma, lista)
print(f'\nResultado final da função reduce: {resultado_final} \n')
  
```

```

Lista original: [1, 2, -3, 4, 5, -9]

>>> Aplicando a função reduce ()
    Somando 1 + 2 = 3
    Somando 3 + -3 = 0
    Somando 0 + 4 = 4
    Somando 4 + 5 = 9
    Somando 9 + -9 = 0

Resultado final da função reduce: 0
  
```


- Entendendo: a função `reduce()` na página anterior, aplica a função `soma()` aos pares dos elementos da lista, dessa maneira temos: $((((1 + 2) + (-3)) + 4) + 5) + (-9) = 0$

Fonte: F. V. C., 2020

Aplicando as funções `reduce()`, list comprehension e `for()`:

```
from functools import reduce

valores = [20, 33, 40, 50, 7, 11, 3, 1, 9, 2, 5]

# reduce com lambda para calcular o produto, cada passo
from functools import reduce

# multiplica todos os valores e apresenta o resultado de cada passo

def calcular_produto(valores):
    # reduce com lambda para multiplicar os valores da lista
    resultados_intermediarios = []
    produto_reduce = reduce(lambda x, y: resultados_intermediarios.append(x * y) or x * y, valores, 1)

    # List comprehension com for para multiplicar os valores
    produto_for, resultados_for = 1, []
    [resultados_for.append(produto_for := produto_for * numero) for numero in valores]

    return produto_reduce, resultados_intermediarios, resultados_for

valores = [20, 33, 40, 50, 7, 11, 3, 1, 9, 2, 5]

produto, resultados_reduce, resultados_for = calcular_produto(valores)

print(f'\n>>> Resultado intermediário usando reduce e lamda:')
print(    resultados_reduce)

print(f'\nResultado final usando reduce e lambda: {produto}\n')

print('>>> Resultado intermediário usando for:')
print(    resultados_for)

print(f'\nResultado final usando for: {resultados_for[-1]}\n')
```

```
>>> Resultado intermediário usando reduce e lamda:
[20, 660, 26400, 1320000, 9240000, 101640000, 304920000, 304920000, 2744280000, 5488560000, 27442800000]

Resultado final usando reduce e lambda: 27442800000

>>> Resultado intermediário usando for:
[20, 660, 26400, 1320000, 9240000, 101640000, 304920000, 304920000, 2744280000, 5488560000, 27442800000]

Resultado final usando for: 27442800000
```

- Guido van Rossum o criador do Python nos alerta para que: só = utilize a função `reduce()` se realmente precisa dela, pois, na maioria dos casos 99 % das vezes uma estrutura de repetição `loop for` é mais legível.

Função all(): é uma função integrada na linguagem de programação Python, projetada para verificar a veracidade de todos os elementos dentro de qualquer iterável (por exemplo, listas, tuplas, conjuntos e dicionários quando usados com chaves). O nome **all** sugere que ela testa se é verdade que **"todos"** os elementos atendem a uma condição verdadeira.

- **Comportamento:** all() retorna **True** se todos os elementos do iterável forem verdadeiros. Um elemento é considerado "verdadeiro" se ele não for igual a False, None, 0, ou qualquer sequência ou coleção vazia ("", (), [], {}), ou qualquer outra instância 'falsy' conforme definido pelas regras de avaliação booleana do Python.

- Retorna **True** se o iterável for vazio, uma vez que não contém elementos que possam ser avaliados como False.

- Retorna False imediatamente se qualquer elemento do iterável for avaliado como False, não prosseguindo com a verificação dos elementos restantes.

- **Casos de uso:**

```
# Exemplo em que todos os elementos são verdadeiros
print(f'\nElementos verdadeiros: {all([1, 2, 3])}')

# Exemplo com um elemento falso
print(f'\nContém pelo menos um elemento falso: {all([1, 0, 3])}')

# Exemplo com iterável vazio
print(f'\nIterável vazio: {all([])} \n')
```

Elementos verdadeiros: True

Contém pelo menos um elemento falso: False

Iterável vazio: True

all() & comprehensions - verificando se há nomes começam com a letra "J"

```
nomes = ["Josefina", "Julieta", "Jurema", "Jucimara",
         "Jacinta", "Jacira", "Joclilde"]

print(f'\nA lista possui nomes que começam com a letra "j"? {all([nome[0] == "J" for nome in nomes])}\n')
```

A lista possui nomes que começam com a letra "j"? True

Fonte: Python Software Foundation. (n.d.). Built-in Functions — Python 3.10.1 documentation. Url: <https://docs.python.org/3/library/functions.html#all>

Função `any()`: uma função **booleana** que retorna **True** se qualquer elemento do iterável for verdadeiro. Se o iterável estiver vazio retorna **False**.

Casos de uso:

```
##### Função any()

lista = [0, 0, 3]

print(f'\nLista original: {lista}')

# Verifica se algum elemento da lista é verdadeiro
print(f'\nA lista contém elementos verdadeiros: {any(lista)}')

# Exemplo com iterável vazio
print(f'\nIterável vazio: {any([])} \n')

# any() & comprehensions - verificando
# há algum elemento que começa com a letra J

nomes = ["Gertrudez", "Jurema", "Jucimara",
         "Jacinta", "Jacira", "Gerimunda"]

print(f'\nLista original: {nomes}')

print(f'\nA lista possui alguns nomes que começam com a letra "J"? {any([nome[0] == "J" for nome in nomes])}\n')
```

Lista original: [0, 0, 3]

A lista contém elementos verdadeiros: True

Iterável vazio: False

Lista original: ['Gertrudez', 'Jurema', 'Jucimara', 'Jacinta', 'Jacira', 'Gerimunda']

A lista possui alguns nomes que começam com a letra "J"? True

Função `isinstance()`, usada para verificar a equivalência de um objeto em Python. Para utilizá-la precisamos importar a **biblioteca `types`**.

- `isinstance()`: verificando a equivalência de um objeto

```
import types

lista = (10, 15, "IFR0")
print(f'\nObjeto original: {lista}')
print(f'\n0 objeto é uma string?: {isinstance(lista, str)}')
print(f'\n0 objeto é um número inteiro? {isinstance(lista, int)}')
print(f'\n0 objeto é um número decimal? {isinstance(lista, float)}')
print(f'\n0 objeto é do tipo booleano? {isinstance(lista, bool)}')
print(f'\n0 objeto é do tipo lista? {isinstance(lista, list)}')
print(f'\n0 objeto é do tipo tupla? {isinstance(lista, tuple)}')
print(f'\n0 objeto é um conjunto? {isinstance(lista, set)}')
print(f'\n0 objeto é um dicionário? {isinstance(lista, dict)}')
```

```
Objeto original: (10, 15, 'IFR0')
0 objeto é uma string?: False
0 objeto é um número inteiro? False
0 objeto é um número decimal? False
0 objeto é do tipo booleano? False
0 objeto é do tipo lista? False
0 objeto é do tipo tupla? True
0 objeto é um conjunto? False
0 objeto é um dicionário? False
```

Exercício_01: utilize *list comprehension* para resolver as seguintes tarefas:

1ª Crie um conjunto chamado `numeros` que contenha todos os números inteiros de 1 a 10.

2ª Números Pares: a partir do conjunto, crie um novo conjunto chamado `pares` que contenha apenas os números pares.

3ª Múltiplos de Três: a partir do conjunto `numeros`, crie um novo conjunto chamado `multiplos_tres` que contenha apenas os múltiplos de três.

Requisitos:

- Utilize apenas *list comprehension* para criar os conjuntos `pares` e `multiplos_tres`.
- Apresente todos os conjuntos criados para verificar seus resultados.

Exercício_02: Crie um dicionário a partir de um *set comprehension*:

- A partir de um conjunto que inicia em 0 (zero) e vai até 9 (nove) criado a partir de *set comprehension*, utilize *dictionary comprehension* para transformar o conjunto em um dicionário.

Tarefas: a partir do conjunto:

1ª Quadrados dos Números: crie um dicionário chamado *quadrados* onde cada chave é um número do conjunto e o valor correspondente é o quadrado desse número.

2ª Números pares e seus quadrados: crie um dicionário chamado *pares_quadrados* que contenha apenas os itens em que as chaves são números pares do conjunto e os valores são os quadrados desses números.

3ª Invertendo número e seu quadrado: a partir do dicionário *quadrados* criado na Tarefa 1, crie um dicionário chamado *inverso* onde as chaves são os quadrados dos números e os valores são os números originais.

Requisitos:

- Utilize apenas *set comprehension* e *dictionary comprehension* para criar os dicionários *quadrados*, *pares_quadrados* e *inverso*.

- Apresente todos os dicionários criados.

Exercício_03 - Nested List Comprehension: Desenvolva uma script que represente de forma simplificada de uma matriz de assentos contendo 20 assentos de um cinema, onde cada linha representa uma fila de assentos e cada assento pode estar disponível (**True**) ou ocupado (**False**).

- Tarefas:

- **1ª Apresente a matriz original**

- **2ª Assentos disponíveis por fila:** Utilize *list comprehension* para criar uma lista chamada *assentos_disponiveis* que contenha o número de assentos disponíveis em cada fila.

- **3ª Assentos ocupados por fila:** Crie uma lista chamada *assentos_ocupados* que, utilizando *nested list comprehension*, contenha todos os índices dos assentos ocupados em cada fila (índices começam em 0).

- **4ª Verificação de fila lotada:** utilizando uma única expressão de *list comprehension*, crie uma lista chamada *filas_lotadas* que contenha os índices das filas que estão totalmente ocupadas.

- Requisitos:

- Para as 2ª e 3ª tarefas, use *list comprehension* para processar a matriz_assentos.
- Para a 4ª tarefa, é permitido usar *list comprehension* externa para gerar a lista, mas a verificação se uma fila está totalmente ocupada deve ser feita com uma *nested list comprehension* interna.
- Apresente todos os conjuntos criados para verificar seus resultados.

Exercício_04: gerenciamento de estoque de produtos

- Desenvolva um script para gerenciar o estoque de produtos de um pequeno comércio eletrônico que vende produtos de papelaria. A lista de estoque deve ser representada por uma lista de tuplas onde cada tupla contém o nome do produto e a quantidade em estoque. Por exemplo: [("produto1", 10), ("produto2", 0), ...] e dada lista representa uma categoria diferente.

- Tarefas:

1ª Apresente a lista de estoques original.

2ª Quantidade total por categoria: utilize map() e lambda para criar uma lista chamada total_por_categoria que contenha a quantidade total de itens por categoria.

3ª Produtos esgotados por categoria: crie uma lista chamada produtos_esgotados que, utilizando *list comprehension*, que contenha todos os nomes dos produtos que estão esgotados (quantidade 0) em cada categoria.

4ª Verificação de categoria esgotada: utilizando map() e uma função com *list comprehension* interna, crie uma lista chamada categorias_esgotadas que contenha os nomes das categorias cujos produtos estão todos esgotados.

- Requisitos:

- Para a 2ª tarefa, use map() e lambda() para processar a lista de estoques.
- Para a 3ª tarefa, use *list comprehension* para identificar produtos esgotados.
- Para a 4ª tarefa, é permitido usar map() com uma função que utilize *list comprehension* internamente para verificar se todos os produtos de uma categoria estão esgotados.
- Para a 5ª tarefa, é permitido usar map() com uma função que utilize *list comprehension* internamente para verificar se todos os produtos de uma categoria não estão esgotados.
- Apresente todos os conjuntos criados para verificar seus resultados.
- **Atenção:** as categorias se referem a esgotados e não esgotados

Persistência de dados: é quando um *script* em desenvolvimento utiliza o armazenamento de dados local ou na nuvem.

Escrevendo em arquivos *Coma Separeted Values* ou *csv*:

Tabela – Modos de abertura de arquivos – serve para arquivos.csv também

Modos	Operações
r	Somente para leitura
w	Escrita, apaga o conteúdo do arquivo
a	Escrita, preserva o conteúdo do arquivo
b	Modo binário
+	Atualização (leitura e escrita)

Fonte: Menezes, 2019.

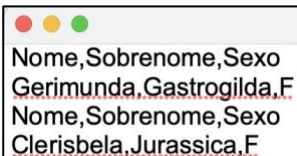
Os modos podem ser combinados ("r+", "w+", "a+", "r+b", "a+b") (MENEZES, 2019, p. 197)

A escrita de dados dentro de um arquivo .csv é feita pela função **writer()** que cria um objeto que nos permite escrever em um arquivo .csv e a função **writerow()** para escrever linha a linha a partir de uma lista.

Escrevendo no arquivo cadastro.csv a partir de uma lista

```
from csv import writer

try:
    with open('cadastro.csv', 'a') as arquivo:
        dados = writer(arquivo)
        nome = ''
        dados.writerow(['Nome', 'Sobrenome', 'Sexo'])
        while nome != 'sair':
            print()
            nome = input('Digite um nome ou sair para encerrar: ')
            if nome != 'sair':
                sobrenome = input('\nDigite o sobrenome: ')
                sexo = input('\nDigite o sexo ("F", "M", "O" : ')
                dados.writerow([nome, sobrenome, sexo.upper()])
except FileNotFoundError:
    with open('cadastro.csv', 'w') as arquivo:
        pass
```



Nome,Sobrenome,Sexo
Gerimunda,Gastrogilda,F
Nome,Sobrenome,Sexo
Clerisbela,Jurassica,F

Refatore o código acima, para que o título das colunas seja gravado apenas uma vez no arquivo, e reorganize o código de acordo com o código da última aula onde desenvolvemos a leitura do arquivo.txt

Exercício_01: Classificação de Números - *dictionary comprehensions*

Dada uma lista de números inteiros: [1, 2, 3, 4, 5], desenvolva um bloco de código que classifique cada número como 'par' ou 'impar' e armazene essas classificações em um dicionário. Para cada número da lista, o número em si será a chave do dicionário, e o valor associado será uma string indicando se ele é 'par' ou 'impar'.

Requisitos:

- O programa deve iniciar com uma lista de números inteiros chamada `numeros`.
- Utilize "**Dictionary Comprehension**" para gerar o dicionário resultante.
- Após a criação do dicionário, imprima-o para que o usuário possa ver a classificação de cada número.

Exercício_2: de *List Comprehension*

Dada uma lista de tuplas: [(4, 3), (1, 2), (8, 9)], em que cada tupla é formada por dois números inteiros, sua tarefa é inverter os elementos dentro de cada tupla e criar uma nova lista de tuplas com os elementos invertidos. O resultado deve ser exibido na tela.

Critérios:

- Utilize a técnica de *list comprehension* para resolver este exercício.
- Não é permitido o uso de laços tradicionais (`for` ou `while`) para este problema.
- Não utilize funções ou bibliotecas externas para inverter a lista.

Atenção: o mecanismo de *list comprehension* é uma forma concisa de criar listas, onde, expressões comuns podem ser substituídas por construções mais curtas e legíveis usando esta técnica. O resultado será: [(3, 4), (2, 1), (9, 8)].

Exercício_3: *dictionary comprehension* – quadrados

- Dado um dicionário com chaves representando letras e valores: `numero = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}`, crie um novo dicionário onde os valores sejam o quadrado dos valores originais. O resultado deve ser: `{'a': 1, 'b': 4, 'c': 9, 'd': 16, 'e': 25}`

Requisitos:

- Utilize a *dictionary comprehension* para resolver o exercício.
- Ao final, imprima o dicionário resultante e o tipo da variável.

Exercício_4: *list comprehension* - tuplas e listas

Dada uma lista de tuplas: [(4, 2, 3), (5, 1, 2), (7, 8, 9)], onde cada tupla contém exatamente três números inteiros, escreva um programa em Python que transforme essa lista em uma nova lista. Nesta nova lista, cada elemento deve ser uma tupla composta pelo primeiro número da tupla original e uma lista com os outros dois números da tupla original. O resultado será: [(4, [2, 3]), (5, [1, 2]), (7, [8, 9])]

Exercício_5: construção de dicionário a partir de sequências

Dada uma string de caracteres únicos: : 'abcde' e uma lista de números inteiros: [1,2,3,4,5], ambos de mesmo comprimento, crie um dicionário onde cada caractere da string se torna uma chave no dicionário e seu valor correspondente deve ser retirado da lista de números. O resultado esperado é: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

Requisitos:

- Não utilize loops tradicionais como for ou while diretamente. Em vez disso, utilize comprehension para criar o dicionário.
- Certifique-se de tratar possíveis erros, como sequências de tamanhos diferentes.
- **Lembre-se** de que o método range() pode ser útil para iterar com base em índices.
- Use a função len() para obter o comprimento de uma sequência.

Exercício_6: list comprehension com raízes quadradas

Dado o módulo *math* que possui uma função *sqrt()* para calcular a raiz quadrada de um número, crie uma lista contendo as raízes quadradas dos números que vão de 3 até 9 (incluindo ambos).

Requisitos:

- Use a técnica de *list comprehension*.
- Nomeie a lista como lista.
- Apresente o resultado.

Lembre-se: a função range(a, b) gera uma sequência de números começando por a e indo até b-1.

Exercício_7: dictionary comprehension com quadrados

Usando a técnica de dictionary comprehension, você precisa criar um dicionário onde as chaves são números inteiros de 1 até 9 (inclusive) e os valores são os quadrados desses números. O resultado é: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

Requisitos:

- O dicionário deve ser nomeado como nrs.
- Utilize a operação de potência (**) para calcular o quadrado de cada número.
- Apresente o dicionário gerado.

Lembre-se: Em Python, $x ** 2$ calcula o quadrado do número x .

Exercício_8: *list comprehension* com raízes quadradas inteiras

Dado o módulo `math` que possui uma função `sqrt()` para calcular a raiz quadrada de um número, crie uma lista contendo apenas os números de 3 até 9 (inclusive ambos) que têm raízes quadradas exatas e inteiras. O resultado é: `[4, 9]`

Requisitos:

- Utilize a técnica de *list comprehension*.
- Nomeie a lista resultante como lista.
- Aprente a lista.
- Para determinar se a raiz quadrada de um número é inteira, lembre-se de que o resto da divisão dessa raiz por 1 deve ser zero.
- Use a operação de módulo (%) para verificar o resto da divisão.

Exercício_9: *dictionary comprehension* com raízes quadradas inteiras

Dado o módulo `math` que fornece uma função `sqrt()` para calcular a raiz quadrada de um número, gere um dicionário onde: as chaves desse dicionário devem ser números inteiros de 3 até 9 (inclusive) que tenham raízes quadradas exatas e inteiras. Os valores associados a essas chaves devem ser suas respectivas raízes quadradas. O resultado é: `{4: 2.0}`.

Requisitos:

- Utilize a técnica de *dictionary comprehension*.
 - Nomeie o dicionário resultante como dicionario.
 - Apresente o resultado na tela.
- Poste todos os códigos e exercícios do material 08-11-2023 em um único arquivo.py na atividade disponibilizada no ambiente.virtua.ifro.edu.br **“Poste todos os códigos e exercícios do material 16-01-2025 em um único arquivo.py até 23-01-2025 às 18:30, não vale pontos.**

Referências

BEAZLEY, D., & Jones, B. K. (2013). **Python Cookbook: Recipes for Mastering Python 3**: Editora: O'Reilly Media. ISBN-10: 1449340377 - ISBN-13: 978-1449340377.

MENEZES, Nilo Ney Coutinho. Introdução à programação com Python. 3ª Edição. 2019. Editora Novatec - São Paulo - SP - Brasil.

MUELLER, John Paul. **Programação Funcional Para Leigos**. Alta Books. Edição do Kindle, 2020.

SWEIGART, A. (2015). **Automate the Boring Stuff with Python: Practical Programming for Total Beginners**. No Starch Press; 1ª edição. ISBN-10: 1593275994, ISBN-13: 978-1593275990.

Boaventura Netto, Paulo Oswaldo. **Grafos, Teoria, modelos, Algoritmos**. Editora Blucher, 2011. São Paulo – SP.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., & STEIN, C. (2009). "**Introduction to Algorithms**" (3ª ed.). MIT Press.

F. V. C. Santos, Rafael. **Python: Guia prático do básico ao avançado (Cientista de dados Livro 2)**. Edição do Kindle - 2019.