CONHECIMENTOS P/ PYTHON

Organizar a ideia melhor sobre como colocar esses comandos

Sumário

1. Introdução / Objetivo

Aqui vou fazer uma introdução para com esse documento e o porque de eu ter criado ele.

2. Métodos Comuns para Strings

2.1. Método len():

O comando len() vai servir para mostrar o tamanho da string. Ele é uma função interna do Python e retorna o comprimento de um objeto, por exemplo, ele pode retornar o número de itens em uma lista. Pode-se usar a função com muitos tipos de dados diferentes.

```
s = "foo"
len(s)
# 3
```

2.2. Métodos upper() e lower():

O método upper() recebe o valor de uma string e retorna ela mesma, mas com todos os caracteres em maiúscula.

E o método lower é justamente o contrário, ele recebe o valor da string e retorna ela mesma com todos os caracteres em minúsculas.

```
>>> frase = 'jogos vorazes'
>>> print(frase.upper()) # TUDO MAIUSCULO
JOGOS VORAZES
>>> print(frase.lower()) # tudo minusculo
jogos vorazes
>>>
```

2.3. Método isalpha():

O método isalpha retorna “True” se todos os caracteres na string forem alfabetos, caso contrário, retorna “False”. Esta função é usada para verificar se o argumento inclui apenas caracteres do alfabeto.

Esse método é bastante utilizado para saber se o que foi digitado (em string/str) foi um número ou não, e se retorna False, ele converte para int.

```
>>> frase1 = 'games'
>>> frase2 = 'alfabeto'
>>> frase3 = '123456'
>>> print(frase1.isalpha())
True
>>> print(frase2.isalpha())
True
>>> print(frase3.isalpha())
False
>>> print('12gts'.isalpha())
False
>>>
```

2.4. **Método isnumeric():**

Isnumeric é um método embutido usado para a manipulação de strings, onde o método retorna “True” se todos os caracteres da string forem numéricos, caso contrário, retorna “False”.

Essa função é usada para verificar se o argumento contém todos os caracteres numéricos, como: inteiros, fações, subscrito, sobrescrito, numerais romanos etc.

```
>>> frase = '123456'
>>> frase2 = 'abc'
>>> frase3 = frase + frase2
>>> print(frase.isnumeric())
True
>>> print(frase2.isnumeric())
False
>>> print(frase3.isnumeric())
False
```

2.5. **Método strip():**

Python é capaz de encontrar espaços em branco dos lados direito e esquerdo de uma string. Strip pega uma cópia da string que tiver esses espaços em branco do início e no final e retorna sem esses espaços.
Também é possível apagar apenas um dos lados (esquerdo ou direito) utilizando o lstrip() - lado esquerdo – e o rstrip() - lado direito

```
>>> frase = ' Jogando o jogo! '
>>> print('>' + frase + '<')
> Jogando o jogo! <
>>> print(frase.lstrip() + '<') # apagar o lado esquerdo
Jogando o jogo! <
>>> print(frase.rstrip() + '<') # apagar o lado direito
 Jogando o jogo!<
>>> print('>' + frase.strip() + '<') # apagar os dois lados
>Jogando o jogo!<
```

2.6. **Método join():**

O método join é usado para especificar os elementos de uma sequência de caracteres para gerar uma nova sequência de conexão. Ou seja, junta cada item da string com um delimitador especificado. É o inverso do split() e também é aceito em listas.

```
>>> frase = 'ABCDEFGHI'
>>> print(','.join(frase))
A,B,C,D,E,F,G,H,I
>>> print(' - '.join(frase))
A - B - C - D - E - F - G - H - I
>>>
```

2.7. **Métodos Split**

O método split é uma das funções disponíveis em Python utilizada para a manipulação de strings. Na prática, ele permite dividir o conteúdo da variável de acordo com as condições especificadas em cada parâmetro da função ou com os valores predefinidos por padrão.

O método split() retorna uma lista de strings após quebrar a string dada pelo separador especificado.

```
>>> frase = 'Aniversario de Fulana'
>>> print(frase.split())
['Aniversario', 'de', 'Fulana']
```

2.8. **Método capitalize():**

O método retorna uma cópia da string original e converte o primeiro caractere da string em uma letra maiúscula enquanto transforma todos os outros caracteres na string em letras minúsculas.

```
>>> frase = 'desenvolvimento em python, show'
>>> frase2 = 'estudos EM sTRING'
>>> print(frase.capitalize())
Desenvolvimento em python, show
>>> print(frase2.capitalize())
Estudos em string
>>>
```

2.9. **Método count():**

Count é uma função embutida na linguagem de programação Python que retorna o número de ocorrências de uma substring na string fornecida. O método count() retorna um inteiro que denota o número de vezes que uma substring ocorre em uma determinada string.

Também a diferenciação de letras maiúscula e minúsculas.

```
>>> frase = 'O rato roueu a roupa do rei de roma'
>>> print(frase.count('r')) # vai contar quantos r tem na frase
5
>>> frase2 = 'Python para Python!'
>>> print(frase.count('Python'))
0
>>> print(frase2.count('Python'))
2
>>> print(frase2.count('python'))
0
```

2.10. **Método replace():**

O método replace(), recebe dois argumentos, para fazer a localização e substituir valores dentro de uma string.

Na sua sintaxe, primeiro vem o argumento antigo e logo depois o novo.

```
>>> frase = 'Musicas, videos, imagens, audios'
>>> print(frase.replace(',','-')) # ',' -> antigo / '-' -> novo
Musicas- videos- imagens- audios
```

2.11. **Método find():**

O método find deve ser utilizado apenas se te interessar a posição da ocorrência na string, isto é, saber em que parte da string foi encontrado o valor desejado. Se a intenção é apenas verificar se for encontrado, o ideal é utilizar o operador in.

Usando o find ele vai retornar o menor índice da primeira aparição dessa substring. Se não encontrar retorna -1.

```
>>> frase = 'Tim Maia foi um bom cantor'
>>> print(frase.find('bom'))
16
>>> print(frase.find('Tim'))
0
>>> print(frase.find('cantor'))
20
```

2.12. **Método title():**

A função title é usada para converter o primeiro caractere em cada palavra em maiúsculas e os caracteres restantes em minúsculas na string e retorna uma nova string.

```
>>> frase = 'a epoca dos anos 50 foi boa!'
>>> print(frase.title())
A Epoca Dos Anos 50 Foi Boa!
>>>
```

3. F-String:

A partir das versões mais recentes do Python, começou-se a usar uma ferramenta para formatação de textos, só que ela não serve apenas e exclusivamente para formatação de textos, ela da mais liberdade para se modificar uma string.

3.1. F-string e suas formatações:

As f-strings vão servir para que você consiga colocar uma variável dentro de um texto, e isso é feito utilizando a letra “f” antes do texto e colocando a sua variável dentro de {} (chaves).

```
>>> frase = 'Temos uma nova frase'
>>> print(f'A frase é: {frase}')
A frase é: Temos uma nova frase
>>>
```

Com isso também nós facilita muito a formatação de outras informações, deixando bem mais fácil nossa vida de programador.

Exemplo de formatação com <u>casas decimais</u>:

```
>>> num = 47.25361235
>>> print(num)
47.25361235
>>> print(f'Numero com 2 casas: {num:.2f}')
Numero com 2 casas: 47.25
>>> print(f'Numero com 4 casas: {num:.4f}')
Numero com 4 casas: 47.2536
```

Exemplo de formatação de <u>Moeda:</u>

```
Formatação de moeda
>>> print(num)
47.25361235
>>> print(f'R$: {num:,.2f}')
R$: 47.25
```

Exemplo de <u>Número fixo de dígitos</u>:

```
Formatação Num Fixo de digitos
>>> celular = 81984523195 # um num de celular tem que ter 11 digitos
>>> celular2 = 81987488 # esse daqui só tem 8 digitos
>>> print(f'Numero: {celular:011}')
Numero: 81984523195
>>> print(f'Numero: {celular2:011}')
Numero: 00081987488
```

3.2. **Método round() no f-string**

Você pode usar a função round() do Python para arredondamento, escolhendo o número de casas que desejar.

A função round() recebe 2 argumentos. O primeiro é o número que deseja arredondar, e o segundo é a quantidade de casas decimais que você deseja.

```
>>> media = 8.055
>>> print(f'Média: {round(media, 2)}')
Média: 8.05
>>> media = 9.9874
>>> print(f'Média: {round(media, 2)}')
Média: 9.99
```

3.3. **Alinhamento a Esquerda, Direita e Centro:**

Alinhamento de texto em Python é útil para imprimir saídas formatas e limpas. Algumas vezes, os dados a serem impressos variam em comprimento, o que os torna confusos quando impressos.

Usando o alinhamento de f-string, a saída pode ser alinhada definindo o alinhamento como esquerdo, direito ou centro e também definindo o espaço (largura) a ser reservado para a string.

Para a sintaxe da string de saída do alinhamento à **esquerda**, defina '<' seguido pelo número da largura.

```
# alinhamento a esquerda
frase = 'Essa casa é maneira'
print('-' * 35)
print(f'{frase:<35}')
print('-' * 35)
```

```
-----------------------------------
Essa casa é maneira
-----------------------------------
```

Para a sintaxe da string de saída de alinhamento à **direita**, defina '>' seguido pelo número da largura.

```
# alinhamento a direita
frase = 'Essa casa é maneira'
print('-' * 35)
print(f'{frase:>35}')
print('-' * 35)
```

```
-----------------------------------
               Essa casa é maneira
-----------------------------------
```

Para a sintaxe da string de saída do alinhamento **central**, defina '^' seguido pelo número da largura

```
# alinhamento no centro
frase = 'Essa casa é maneira'
print('-' * 35)
print(f'{frase:^35}')
print('-' * 35)
```

```
-----------------------------------
        Essa casa é maneira
-----------------------------------
```

Exemplo com todos os 3:

```
#Exemplo com os 3 alinhamentos
frase = 'Eu lamento - Tim maia'
print('-' * 35)
print(f'{frase:^35}')
print('-' * 35)
print(f'{frase:<35}')
print('-' * 35)
print(f'{frase:>35}')
print('-' * 35)
```

```
-----------------------------------
       Eu lamento - Tim maia
-----------------------------------
Eu lamento - Tim maia
-----------------------------------
              Eu lamento - Tim maia
-----------------------------------
```

4. Fatiamento de Strings

O fatiamento consiste em obter uma sub-string de uma determinada string, dividindo-a respectivamente do início ao fim. Selecionar uma fatia é como selecionar um caractere.

Para fatiar uma string, precisamos entender que em toda string a uma sequência. Dentro de uma string a índices de determinada sequência a partir do zero.

Fatiamento de string é muito utilizado em tuplas, listas e dicionários.

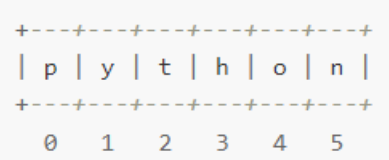


Ilustração 1: Exemplo de sequência com a string "python".

Outro coisa que precisamos saber sobre fatiamento de sequências é:

- Ao fatiar uma string teremos como resultado uma nova string fatiada;
- Ao fatiar uma lista teremos como resultado uma nova lista com os “pedaços”.

4.1. Como fatiar em python

Dentro de python, a utilização dos índices que vimos anteriormente vai nos auxiliar para conseguir ‘capturar’ a sub-string que queremos.

Vamos utilizar o exemplo da ilustração 1, eu quero pegar apenas as letras do índice 3 até o 5.

```
>>> frase = 'python'
>>> print(frase[3:5])
ho
>>>
```

Lembrando que, em python ele vai pegar até um número antes. Então ele pegou apenas as letras ‘h’ e ‘o’.

O operador **[n:m]** retorna a parte da string do “enésimo” caractere ao “emésimo” caractere, incluindo o primeiro, mas excluindo o último. Este comportamento é contraintuitivo, porém pode ajudar a imaginar os índices que indicam a parte entre os caracteres.

Obs.: Se o primeiro índice for maior ou igual ao segundo, o resultado é uma string vazia, representada por duas aspas

```
>>> frase = 'Thiago Luiz'
>>> print(frase[5:])
o Luiz
>>> print(frase[0:5])
Thiag
>>> print(frase[:])
Thiago Luiz
>>> print(frase[3:3])

>>>
```

4.2. **Percorrer uma fatia com um laço**

Você pode usar uma fatia em uma laço for se quiser percorrer um subconjunto de elementos de uma lista.

```
>>> players = ['charles', 'martina', 'michael', 'florence']
>>> for player in players[:3]:
...     print(player.title())
...
Charles
Martina
Michael
>>>
```

As fatias são muito úteis em várias situações. Por exemplo, ao trabalhar com dados, você pode usar fatias para processar seus dados em porções de tamanho específico.

Ou então, quando criar um jogo, você poderia adicionar a pontuação final de um jogador em uma lista sempre que esse jogador acabar de jogar. Seria possível então obter as três pontuações mais altas de um jogador ordenando a lista em ordem decrescente e obtendo uma lista que inclua apenas as três primeiras pontuações.

5. Utilizações em Listas

5.1. **Método Sort()** → falta finalizar

Esse método é usado para ordenar uma lista diretamente, o que significa que ela realiza a mutação da lista ou que a modifica diretamente sem criar cópias adicionais. Ela pode ordenar uma lista em ordem ascendente ou ordem descendente.

Quando o método sort() é utilizado ele modifica a lista e, portanto, sua versão original se perde. Por causa disso, você somente deve usar esse método se:

- Quiser modificar (ordenar) a lista permanentemente;

- Não precisar manter a versão original da lista.

A sintaxe do método é bem simples: *<list>.sort()*

É possível passar argumentos, mas caso você não passe, por padrão:

- A lista será ordenada em ordem ascendente;
- Os elementos da lista serão comparados diretamente usando seus valores com o operador <.

Exemplo:

```
>>> bv = [9,8,7,6,5,4,3,2,1,0]
...  #exemplo de lista
>>> bv.sort()
>>> print(bv)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

5.2. **Copiando uma lista**

Com frequência, você vai querer começar com uma lista existente e criar uma lista totalmente nova com base na primeira.

Para criar uma lista, podemos criar uma fatia que inclua a lista original inteira omitindo o primeiro e o segundo índices ([:]). Isso diz a Python para criar uma lista que começa no primeiro item e termina no último, gerando uma cópia da lista toda.

```
>>> print(filmes)
    ['Jogos vorazes', 'Panico', 'O juiz', 'Arquivo-X', 'O Zorro']
>>> filmes_v2 = filmes[:]
>>> print(filmes)
    ['Jogos vorazes', 'Panico', 'O juiz', 'Arquivo-X', 'O Zorro']
>>> print(filmes_v2)
    ['Jogos vorazes', 'Panico', 'O juiz', 'Arquivo-X', 'O Zorro']
>>>
```

5.3. **Transferindo itens de uma lista para outra (pop e append)**

Considere uma lista de usuários recém-registrados em um site, porém não verificados. Depois de conferir esses usuários, como podemos transferi-los para uma lista separada de usuários confirmados? Uma maneira seria usar um laço while para extrair os usuários da lista de usuários não confirmados à medida que os verificarmos e então adicioná-los em uma lista separada de usuários confirmados.

```python
# Começa com os usuários que precisam ser verificados,
# e com uma lista vazia para armazenar os usuários confirmados
users_nao_confirmado = ['Alice', 'Jonathan', 'Gustavo', 'Caroline']
users_confirmado = []

# O while verifica cada usuário até que não haja mais usuários não confirmados
# E transfere cada usuário verificado para a lista de usuários confirmados.
while users_nao_confirmado:
  current_user = users_nao_confirmado.pop()
  print(f'Usuário verificado: {current_user.title()}')
  users_confirmado.append(current_user)

# o for exibe todos os usuários confirmados
print('\nOs usuários confirmados são: ')
for user_confirmado in users_confirmado:
  print(user_confirmado.title())
```

```
Usuário verificado: Caroline
Usuário verificado: Gustavo
Usuário verificado: Jonathan
Usuário verificado: Alice

Os usuários confirmados são:
Caroline
Gustavo
Jonathan
Alice
```

O laço while vai executar o looping enquanto a lista *users_nao_confirmado* não estiver vazia. Nesse laço, a função pop() remove os usuários não verificados, um de cada vez, do final de *users_nao_confirmado*. Nesse caso, como Caroline é o último elemento da lista *users_nao_confirmado*, sei nome será o primeiro a ser removido, armazenando em *current_user* e adicionando à lista *users_confirmado*. Assim segue-se com os outros elementos dentro da lista.

Quando a lista de usuários não confirmados estiver vazia, o laço para e a lista de usuários confirmados é exibida.

5.4. **Removendo instâncias de uma lista (remove)**

Assim como é possível adicionar instâncias dentro de uma lista, também é capaz de removê-las. O método remove() é bem simples de se usar, como veremos.

```python
cores = ['azul', 'cadeira','branco', 'preto', 'cadeira']
cores.remove('cadeira')
print(cores)
```

```
['azul', 'branco', 'preto', 'cadeira']
```

Contudo, caso haja mais de uma instância com o mesmo nome, o comando remove apenas apagará uma única vez. Como podemos ver no exemplo, dentro da lista *cores* havia duas instâncias com o nome cadeira, uma no índice 1 e a outra no índice 4. Mas, quando foi utilizado o método remove(), apagou-se apenas o que apareceu primeiro.

Para apagar todas as instâncias que forem repetidas, basta colocar um laço while até que a instância não esteja mais na lista.

```
moveis = ['cadeira', 'mesa', 'sofa', 'estante', 'mesa']
print(moveis)
while 'mesa' in moveis:
  moveis.remove('mesa')
print(moveis)
```

```
['cadeira', 'mesa', 'sofa', 'estante', 'mesa']
['cadeira', 'sofa', 'estante']
```

5.5. **Passando uma lista para uma função**

Com frequência, você achará útil passar uma lista para uma função, seja uma lista de nomes, de números ou de objetos mais complexos, como dicionários. Se passarmos uma lista a uma função, ela terá acesso direto ao conteúdo dessa lista.

Suponha que tenhamos uma lista de usuários e queremos exibir uma saudação a cada um. O exemplo a seguir envia uma lista de nomes a uma função chamada saudacoes(), que saúda cada pessoa da lista individualmente.

```
def saudacoes(nome):
  for n in nome:
    msg = f'Olá, {n.title()}!'
    print(msg)
usuarios = ['Sebastiana', 'Marcos', 'Vitoria']
saudacoes(usuarios)
```

```
Olá, Sebastiana!
Olá, Marcos!
Olá, Vitoria!
```

6. Utilizações em Tuplas

7. Utilizações em Dicionários

Dicionários são capazes de armazenar uma quantidade quase ilimitada de informações, mostraremos como percorrer os dados de um dicionário com um laço. Entender os dicionários permite modelar uma diversidade de objetos do mundo real de modo mais preciso.

Um dicionário em Python é uma coleção de pares chave-valor. Cada chave é conectada a um valor, e você pode usar uma chave para acessar o valor associado a ela. O valor de uma chave pode ser um número, uma string, uma lista ou até mesmo outro dicionário.

Um dicionário é representado entre **chaves {}**, com uma série de pares chave-valor entre elas.

7.1. **Modificando valores em um dicionário**

Para modificar um valor em um dicionário, especifique o nome do dicionário com a chave entre colchetes e o novo valor que você quer associar a essa chave.

Como um exemplo, imagine um carro da cor prata, que depois da pintura ficou com a cor preta;

```
>>> car = {'ano':2012, 'cor':'prata'}
>>> print(f'A cor do carro é {car["cor"]}')
A cor do carro é prata
>>> car['cor'] = 'preto'
>>> print(f'A cor do carro depois da pintura é {car["cor"]}')
A cor do carro depois da pintura é preto
>>>
```

7.2. **Percorrendo um dicionário com um laço**

Um único dicionário python pode conter apenas alguns pares chave-valor ou milhões deles. Como um dicionário pode conter uma grande quantidade de dados, python permite percorrer um dicionário com um laço.
Dicionários podem ser usados para armazenar informações de várias maneiras; assim, há diversos modos diferentes de percorrê-los com um laço.

```
users = {'username':'Paulo',
         'first':'Victor',
         'last':'Silva'}
for key, value in users.items():
  # variáveis key e value podem ser qualquer
  # uma. Isso é só um exemplo
  print(f'\nKey: {key}')
  print(f'Value: {value}')
```

7.2.1. Percorrendo todas as chaves com um laço

O método keys() é conveniente quando não precisamos trabalhar com todos os valores de um dicionário.

```
comida_fav = {'José':'Pizza',
              'Matias':'Sanduiche',
              'Naldo':'Batata Frita',
              'Maira':'Macarronada'}
# for pessoas in comida_fav:  ou \/
for pessoas in comida_fav.keys():
  print(pessoas.title())
  # A saída vai ser apenas os nomes
  # das pessoas do dicionário!
```

O método keys() não serve apenas para laços: na verdade, ele devolve uma lista de todas as chaves.

7.2.2. Percorrendo todos os valores com um laço com values() e set()

Se você estiver ais interessando nos valores contidos em um dicionário, o método values() pode ser usado para devolver uma lista de valores sem as chaves.

```
comida_fav = {'José':'Pizza',
              'Matias':'Sanduiche',
              'Naldo':'Batata Frita',
              'Maira':'Macarronada'}
for comida in comida_fav.values():
  print(comida.title())
  # A saída vai ser apenas as comidas
  # favoritas do dicionário!
```

Essa abordagem com o método values() extrai todos os valores do dicionário, sem verificar se há repetições. Isso pode funcionar bem com uma quantidade pequena de valores, mas imagine o exemplo anterior, contendo repetições de comidas favoritas, o resultado seria uma lista com muitas repetições.

Para ver cada valor sem repetições, podemos usar um conjunto (set). Um conjunto é semelhante a uma lista, exceto que cada item de um conjunto deve ser único.

```
comida_fav = {'José':'Pizza',
              'Matias':'Sanduiche',
              'Naldo':'Batata Frita',
              'Maira':'Macarronada',
              'Eduarda':'Pizza'}
for comida in comida_fav.values():
  print(comida.title())
  # Sem o comando set() o resultado sairá
  # com repetições
```

```
Pizza
Sanduiche
Batata Frita
Macarronada
Pizza
```

*Percorrendo valores sem o método **set()***

```
comida_fav = {'José':'Pizza',
              'Matias':'Sanduiche',
              'Naldo':'Batata Frita',
              'Maira':'Macarronada',
              'Eduarda':'Pizza'}
for comida in set(comida_fav.values()):
  print(comida.title())
  # Sem o comando set() o resultado sairá
  # com repetições
```

```
Macarronada
Pizza
Sanduiche
Batata Frita
```

*Percorrendo valores **com** o método **set()***

Quando colocamos set() em torno de uma lista que contenha itens duplicados, Python identifica os itens únicos na lista e cria um conjunto a partir desses itens.

7.3. **Preenchendo um dicionário com input**

Como já é ciente, também é possível criar-se um dicionário em que nele, as chaves e valores sejam o próprio usuário que digite.

Vamos criar um exemplo em que haja uma enquete em que solicita ao usuário o nome e uma resposta. Armazenaremos os dados coletados em um dicionário, pois queremos associar cada resposta a um usuário em particular

```
respostas = {}
votacao_ativa = True # Essa flag vai indicar que a enquete está ativa
while votacao_ativa:
  nome = input('Qual seu nome? ').title()
  lugar = input('Diga um lugar que deseja visitar? ').title()
  respostas[nome] = lugar # Armazena a resposta no dicionário
  repetir = input('Outra pessoa deseja reponder a enquete [S/N]? ').upper()
  # vai servir para descobrir se outra pessoa vai responder à enquete

  if repetir == 'N':
    votacao_ativa = False

print('\n-- Resultados --')
for n, l in respostas.items():
  print(f'{n} quer visitar {l}')
```

```
Qual seu nome? joao
Diga um lugar que deseja visitar? paris
Outra pessoa deseja reponder a enquete [S/N]? s
Qual seu nome? marcos
Diga um lugar que deseja visitar? venezuela
Outra pessoa deseja reponder a enquete [S/N]? n

-- Resultados --
Joao quer visitar Paris
Marcos quer visitar Venezuela
```

8. Informações Aninhadas

Às vezes, você vai querer armazenar um conjunto de dicionários em uma lista ou uma lista de itens como um valor em um dicionário. Isso é conhecido como aninhar informações. Podemos aninhar um conjunto de dicionários em uma lista, uma lista de itens em um dicionário ou até mesmo um dicionário em outro dicionário. Aninhar informações é um recurso eficaz.

8.1. **Uma lista de dicionários**

É comum armazenar vários dicionários em uma lista quando cada dicionário tiver diversos tipos de informação sobre um objeto. Todos os dicionários da lista devem ter uma

estrutura idêntica para que possamos percorrer a lista com um laço e trabalhar com cada objeto representado por um dicionário do mesmo modo.

Vamos imaginar uma garagem de um prédio com mais de 10 carros. Um dicionário contém várias informações sobre um só carro, mas não haveria como adicionar mais informações sobre um segundo modelo de carro.
Uma maneira de contornar isso, é criar uma lista de carros, em que cada carro seja representado por um dicionário com informações sobre ele.

```
# nesse exemplo, será só com três carros
carro1 = {'Marca':'Ford', 'Ano':'2012', 'Modelo':'Ka', 'AP':'12'}
carro2 = {'Marca':'Fiat', 'Ano':'2015', 'Modelo':'Palio', 'AP':'100'}
carro3 = {'Marca':'Nissan', 'Ano':'2012', 'Modelo':'Sentra', 'AP':'213'}

garagem = [carro1, carro2, carro3]

for car in garagem:
  print(car)
```

```
{'Marca': 'Ford', 'Ano': '2012', 'Modelo': 'Ka', 'AP': '12'}
{'Marca': 'Fiat', 'Ano': '2015', 'Modelo': 'Palio', 'AP': '100'}
{'Marca': 'Nissan', 'Ano': '2012', 'Modelo': 'Sentra', 'AP': '213'}
```

8.2. **Uma lista em um dicionário**

Em vez de colocar um dicionário em uma lista, às vezes é conveniente colocar uma lista em um dicionário. Vamos usar um exemplo de que alguém está pedindo uma pizza. Se usássemos apenas uma lista, tudo que poderíamos realmente armazenar é uma lista dos ingredientes da pizza. Com um dicionário, uma lista de ingredientes pode ser apenas um dos aspectos da pizza que estamos descrevendo.

```
pizza = {'Borda':'fina',
         'coberturas':['Queijo', 'Frango']}
print(f'Você pediu uma pizza com borda {pizza["Borda"]},')
print('Com os seguintes ingredientes: ')
for cober in pizza['coberturas']:
  print(f'\t{cober}')
```

```
Você pediu uma pizza com borda fina,
Com os seguintes ingredientes:
        Queijo
        Frango
```

8.3. **Um dicionário em um dicionário**

Podemos aninhar um dicionário em outro dicionário, mas o código poderá ficar complicado rapidamente se isso for feito.
Por exemplo, se você tiver vários usuários em um site, cada um com um nome único, os nomes dos usuários poderão ser usados como chaves em um dicionário. Você poderá então armazenar informações sobre cada usuário usando um dicionário como o valor associado a cada nome de usuário.

```
users = {
      'aeinstein':{'first':'Albert', 'last':'Einstein', 'location':'Princeton'},
        'mcurie':{'first':'Marie', 'last':'Curie', 'location':'Paris'}}
for username, user_info in users.items():
  print(f'\nUsername: {username}')
  Nome_completo = f"{user_info['first']} {user_info['last']}"
  location = user_info['location']
  print(f'\tNome Completo: {Nome_completo.title()}')
  print(f'\tLocalização: {location.title()}')
```

```
Username: aeinstein
        Nome Completo: Albert Einstein
        Localização: Princeton

Username: mcurie
        Nome Completo: Marie Curie
        Localização: Paris
```

9. Módulos em Python

Importar um módulo em seu programa principal é dizer ao Python para deixar o código de seu módulo disponível no arquivo de programa em execução no momento. Alguns desses módulos (ou bibliotecas) são disponíveis por padrão quando instalamos o Pyhton, outros temos que instalar manualmente.

9.1. Importando uma função (própria) para um módulo

Uma vantagem das funções é a maneira como elas separam blocos de código de seu programa principal. Ao usar nomes descritivos para suas funções, será bem mais fácil entender o seu programa principal. Você pode dar um passo além armazenando suas funções em um arquivo separado chamado módulo e, então, importar esse módulo em seu programa principal.

Armazenar suas funções em um arquivo separado permite ocultar os detalhes do código de seu programa e se concentrar na lógica de nível mais alto. Também é permitido reutilizar funções em muitos programas diferentes.

Quando armazenamos funções em arquivos separados, podemos compartilhar esses arquivos com outros programadores sem a necessidade de compartilhar o programa todo.

Para começar a importar funções, inicialmente precisamos criar um módulo. Um módulo é um arquivo terminado em *.py* que contém o código que queremos importar para o nosso programa.

Para nosso exemplo, criaremos um módulo chamado *interface.py*. Que nele conterá:

```
def cabecalho(msg):
  tam = len(msg)
  print('+', '=' * tam, '+')
  print('|', msg, '|')
  print('+', '=' * tam, '+')
```

Agora criaremos um arquivo separado, eu chamarei

de *programa_principal.py* no mesmo diretório que este *interface.py*.

Nesse arquivo, importaremos o módulo que acabamos de criar e, em seguida, faremos duas chamadas para apresentar um cabeçalho para nossa mensagem.

```python
import interface  # <- Aqui importamos o módulo
# só 'import interface' fará com que todas as
# funções do módulo seja importado;
# caso queira apenas um, digite:
# from intercafe import cabecalho

interface.cabecalho('Jogar é bom demais!')
interface.cabecalho('Vamos ler um pouco')
```

```
+ ================== +
| Jogar é bom demais! |
+ ================== +
+ ================== +
| Vamos ler um pouco |
+ ================== +
```

9.2. **Módulo Random**

10. Cores em Python

11. Tratamento de erros e Exceções

12. Funções para validação de dados

Enquanto estudava sobre funções em python, me deparei com um exercício no qual o foco era simplesmente a validação de dados. Em python, e com o meu pouco conhecimento que tenho até agora, sei que existem diversas maneiras de se fazer uma validação. Contudo, alguns desses métodos foram maneiras interessantes que eu vim a conhecer, então caso eu encontre outras maneiras de verificar dados, irei complementar nessa seção. Como também complementarei com maneira de como validar outros tipos

de dados como CPF, E-mail, Número de telefone etc.

12.1. **LeiaInt()**

O método leiaInt não está na biblioteca padrão do Python, mas bem que poderia estar, já que facilita e muito nossa vida na hora de validar um dado número para inteiro. Esse método eu vi no exercício 104 do canal do Curso em Vídeo. Todos os créditos vão a ele.

```
def leiaInt(msg):
    ok = False  # variável puramente para encerrar o laço
    valor = 0  # variável que armazenará o valor se ele for numerico, no final.
    while True:
        n = str(input(msg))  # n vai servir como uma variável qualquer
        if n.isnumeric():  # Se, a variável n tiver valores numericos, então o programa fará:
            valor = int(n)  # a variável 'valor' vai receber n como inteiro
            ok = True
        else:
            print('Você não digitou um número. Tente Novamente!')

        if ok:
            break
    return valor
```

12.2. **valida_int()**

Esse método eu aprendi na disciplina de Lógica de programação e Algoritmos do curso de Análise e desenvolvimento de sistemas, na aula 05, que falava sobre funções. Esse método não é melhor que o anterior (leiaint), mas dependendo do caso, ele pode ser bem proveitoso e, talvez, melhor.

O valida_int recebe três (3) parâmetros, sendo eles:

1º → um input que recebe a string;

2º → um valor mínimo;

3º → um valor máximo.

```
def valida_int(pergunta, min, max):
    x = int(input(pergunta))
    while (x < min) or (x > max):
        x = int(input(pergunta))
    return x
```

12.3. **Validar CPF**

13. Orientação a Objetos

Como os estudos de Orientação a Objetos (OO) é um assunto bem extenso e de ‘importância’ para programadores, incluindo principalmente eu, decidi incluir um tópico nesse documento, para anotar meus estudos e também para caso no futuro venha a esquecer sobre o que poderia vir a ser os conceitos iniciais da orientação a objetos. Como meu curso de graduação não adentrou ainda ao assunto nos momentos iniciais desse capítulo, eu estou utilizando diversas fontes da internet, como livros, vídeos e até ajuda dos meus colegas de classe. A você que está lendo (podendo ser eu ou, caso eu coloque no git esse arquivo, outra pessoa) espero que possa ajudar a entender nem que seja o básico sobre OO.

13.1. O que é Programação Orientada a Objetos?

A orientação a objetos é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.

Embora a Orientação a Objetos tenha vantagens em relação aos paradigmas que a precederam, existe uma desvantagem inicial: ser um modo mais complexo e difícil de se pensar.

Isso pode ser atribuído à grande quantidade de conceitos que devem ser assimilados para podermos trabalhar orientado a objetos. Isso pode ser atribuído à grande quantidade de conceitos que devem ser assimilados para podermos trabalhar orientado a objetos.

13.2. Classes

Embora POO baseia-se fundamentalmente no termo “Objeto”, tudo começa com a definição do que é uma classe. Antes mesmo de ser possível manipular objetos, é preciso definir uma classe, pois esta é a unidade inicial e mínima de código na OO.

Classe é uma estrutura que abstrai um conjunto de objetos características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar.

O objetivo de uma classe é definir, servir de base, para o que futuramente será o objetivo. É através dela que criamos o “molde” aos quais os objetos deverão seguir.

Uma classe também pode ser definida como uma abstração de uma entidade, seja ela física ou conceitual do mundo real. É através de criação de classes que se conseguirá codificar todas as necessidades de um sistema (software).

Outro fator importante para classes é o seu nome. O nome deve representar bem sua finalidade dentro do contexto ao qual ela foi necessária. Vamos criar um exemplo:

→ Em um sistema de controle de restaurante, temos uma classe chamada *pessoa*, essa classe vai representar todos os indivíduos dali, como clientes, garções, cozinheiros, recepcionistas, entregadores etc.

13.3. O Atributo

Só definir uma classe não é suficiente para começar a manipulá-las. Devemos também detalhá-las para especificar suas características e o que vai ser colocado ali dentro. Essas características é que vão definir quais informações as classes poderão armazenar e manipular. Na OO estas características e informações são denominadas de atributo.

Atributo é o elemento de uma classe, responsável por definir sua estrutura de dados. O conjunto destes será responsável por representar suas características e farão parte dos objetos criados a partir da classe.

Essa definição deixa bem claro que os atributos devem ser definidos dentro da classe. Devido a isso, são responsáveis por definir sua estrutura de dados. É a partir do uso de atributos que será possível caracterizar (detalhar) as classes, sendo possível representar fielmente uma entidade do mundo real.

Há dois tipos principais de atributos:

- **Estáticos**: Mantém o mesmo valor durante sua existência, como a data de nascimento de uma pessoa;
- **Dinâmicos**: Valores que variam com o passar do tempo, como a idade de uma pessoa.

→ Pegando o exemplo anterior de um restaurante, dentro da classe *Pessoa* podemos criar atributos para elas. Como:

– Tipo (cliente, funcionário), nome, idade, gênero, etc.

13.4. O Objeto

Já tínhamos visto que tudo começa com a definição de uma classe. Então o que é um

objeto? É a instanciação de uma classe.

Um objeto é a representação de um conceito/entidade do mundo real, que pode ser física (bola, carro, árvore etc.) ou conceitual (viagem, estoque, compra etc.) e possui um significado bem definido para um determinado software. Para este conceito/entidade, deve ser definida inicialmente uma classe a partir da qual posteriormente serão instanciados objetos distintos.

Por definição, todo objeto é único. Se tivermos uma classe e forem criados dois objetos a partir dela, cada um será diferente do outro, mesmo que seus estados sejam iguais por coincidência.

Objetos tem características próprias (*atributos*) e executa ações (*métodos*), provenientes da classe que originou o objeto.

13.5. A Mensagem

Mensagem é o processo de ativação de um método de um objeto. Isto ocorre quando uma requisição (chamada) a esse método é realizada, assim disparando a execução de seu comportamento descrito por sua classe. Pode também ser direcionada diretamente a classe, caso a requisição seja a um método estático.

A definição anterior deixa bem claro que, quando requisitamos que um comportamento (código) de um método seja executado, estamos passando uma mensagem a este método. Mensagens podem ser trocadas entre métodos dos objetos/classes, para serem realizadas as atividades inerentes a cada um.

13.6. Método

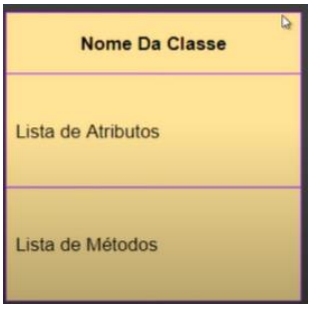
Método é uma lógica contida em uma classe para atribuir comportamentos (sequência de comandos), identificada por um nome. É muito similar a funções e procedimentos. O ato de invocar (chamar) um método é a passagem de mensagens para o objeto.

Uma função dentro da classe vai se chamar método!

13.7. O que é UML?

É uma linguagem gráfica de modelagem de sistemas orientados a objetos. É um padrão mundialmente aceito. UML significa *Unified Modeling Language* (em português, Linguagem de Modelagem Unificada).

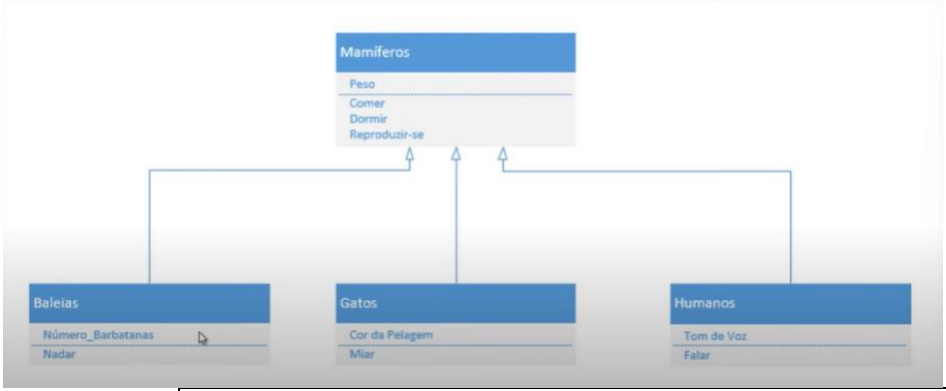
Ela provê várias notações gráficas para facilitar o processo de modelagem de sistemas OO. A forma padrão para representação de uma classe usando um diagrama de classes UML é:



13.8. **Herança**

Herança é o relacionamento entre classes em que uma classe chamada de subclasse (classe filha, classe derivada) é uma extensão, um subtipo, de outra classe chamada de superclasse (classe pai, classe mãe, classe base). Devido a isto, a subclasse consegue reaproveitar os atributos e métodos dela. Além dos que venham a ser herdados, a subclasse pode definir seus próprios membros.

Em outras palavras, herança é um relacionamento entre classes, no qual uma classe “herda” os membros (atributos e métodos) de outra classe. Assim, podemos criar classes mais complexas sem repetição de código.



→ Vamos criar um exemplo em que precisamos classificar mamíferos no reino animal.

→ Vamos pegar os mamíferos: Homens, Baleias e Gatos. Esses compartilham características, mas possuem atributos distintos. Como poderíamos modelar eles?

→ Podemos criar classes distintas para cada ser, mas essas classes teriam muitos comportamentos em comum, o que ocasionaria repetição desnecessária de código.

→ A herança resolve esse problema:

Criamos uma classe **Mamífero** que possua as características comuns a todos os mamíferos, e então as classes Humano, Baleia e Gato herdando essas características e também implementando as funcionalidades específicas de cada animal, como Falar, Andar, Nadas e Miar.

A seta sempre aponta da classe ‘filho’(*que está herdando)* para a classe ‘pai’ (*que está com suas características sendo herdadas*).

13.9. Polimorfismo

Em determinados momentos em uma hierarquia de classes, precisamos que um mesmo método (nome e lista de parâmetro, ou seja, assinatura) se comporte de forma diferente dependendo do objeto instanciado a partir de uma classe de uma hierarquia qualquer.

Uma operação de um objeto pode assumir mais de um comportamento dependendo da chamada recebida, tratando e devolvendo respostas distintas.
Polimorfismo ocorre quando um objeto tem um comportamento diferente para uma mesma ação.

Poli* =** Muitos; Morphos*** = Formas → Portanto, polimorfismo significa “muitas formas”.

→ Exemplo: nas classes Gato e Humano há um comportamento (método) em comum: **Andar**.

→ Porém, cada uma dessas classes implementa o método de forma distinta – gatos andam em quatro patas, e os humanos andam em duas pernas.

→ Desta forma, dependendo de como um método é invocado, seu comportamento será diferente.

13.10. Definir uma classe em Python

A sintaxe simplificada para criar uma classe dentro é python é a seguinte:

```
class NomeClasse: # cabeçalho da classe
  pass # classe sem função (ainda)

print(type(NomeClasse))
# ver o tipo da classe
```

Dentro da classe se pode então inicializar com o método __init__

13.10.1. Introdução ao Método __init__

Esse método __init__, também conhecido como método especial, é o construtor da classe.

Como o nome sugere, ele será o responsável por criar objetos a partir da classe em questão. Ou seja, sempre que for necessário criar objetos de uma determinada classe, seu construtor deverá ser utilizado.

Este método define o construtor da classe, como já foi falado, e nada mais é de onde definimos com uma nova instância será criada.

> → Um exemplo para isso, é um registro de clientes em um restaurante. Onde cada cliente que entra precisa se registrar.
>
> → Então trazendo pra python: toda vez que a classe Cliente é iniciada o __init__ faz seu papel para construir uma nova instância.

13.10.1.1. Método __init__() de uma subclasse (herança)

Como já vimos anteriormente, uma herança é quando uma classe herda de outra, assumindo automaticamente todos os atributos e métodos da primeira classe.

A classe original se chama *classe-pai* e a nova é a *classe-filha.*

A primeira tarefa de Python ao criar uma instância de uma classe-filha é atribuir valores a todos os atributos da classe-pai. Para isso, o método __init__() de uma classe-filha precisa da ajuda de sua classe-pai.

13.10.2. Entendendo o self em classes

O uso do **self** é usado em classes no Python para indicar que você está referenciando alguma coisa do próprio objeto (sejam eles atributos ou métodos) – na verdade, o self é o próprio objeto em si.

Em geral, quando criamos um construtor de uma classe (com o método __init__) colocamos o self nos atributos definidos ali para que eles sejam acessíveis nos métodos (raramente uma tributo que você inicializa no construtor não será usado em algum canto na classe).

Resumindo, todo método de uma classe recebe como primeiro parâmetro uma referência à instância que chama o método (isso é uma obrigação), permitindo assim que o objeto acesse os seus próprios atributos e métodos. Por convenção, chamamos esse primeiro parâmetro de self, mas qualquer outro nome poderia ser utilizado.

13.11. Exemplos Práticos em Python

=> EX1: Para iniciar, criaremos uma classe simples que pegará informações de um cachorro e retornará nos prints uma frase.

```
# Exemplo 1

class Cao:
  def __init__(self, nome, raca, idade):
    self.nome = nome
    self.raca = raca
    self.idade = idade

# Características do Primeiro Cachorro
meu_cao = Cao('Zake', 'Poodle', 1)
print(f'O primeiro meu chachorro {meu_cao.nome.title()} é da ', end='')
print(f'raça {meu_cao.raca} e tem {meu_cao.idade} ano!')

# Cacracterísticas do Segundo Cachorro
meu_cao2 = Cao('Thor', 'Pitcher', 3)
print(f'O meu segundo chachorro {meu_cao2.nome.title()} é da ', end='')
print(f'raça {meu_cao2.raca} e tem {meu_cao2.idade} anos!')
```

```
O primeiro meu chachorro Zake é da raça Poodle e tem 1 ano!
O meu segundo chachorro Thor é da raça Pitcher e tem 3 anos!
```

=> EX2: Vamos escrever uma nova classe que represente um carro. Nossa classe armazenará informações sobre o tipo de carro com que estamos trabalhando e terá um método que sintetiza essa informação.

```
# Exemplo 2

class Carro:
  # Tentativa simples de representar um carro!
  def __init__(self, fabricante, modelo, ano):
    # Vai inicializar os atributos que descrevem um carro!
    self.fabricante = fabricante
    self.modelo = modelo
    self.ano = ano

  def descricao(self):
    # Devolve um nome descritivo, formatado de modo elegante.
    nome_longo = f'{str(self.ano)} {self.fabricante} {self.modelo}'
    return nome_longo.title()

novo_carro = Carro('audi', 'a4', 2016)
print(novo_carro.descricao())
```

```
2016 Audi A4
```

=> EX3: Vamos pegar o exemplo anterior de criar uma classe Carro, e deixar mais interessante, que é adicionar um atributo que mude com o tempo. Acrescentaremos um atributo que armazena a milhagem (KM) do carro.

```python
# Exemplo 3

class Carro:
  # Tentativa simples de representar um carro!
  def __init__(self, fabricante, modelo, ano):
    # Vai inicializar os atributos que descrevem um carro!
    self.fabricante = fabricante
    self.modelo = modelo
    self.ano = ano
    self.kilometragem = 0  # instância para ler os KM
    # Essa instância também tem um valor default!

  def descricao(self):
    # Devolve um nome descritivo, formatado de modo elegante.
    nome_longo = f'{str(self.ano)} {self.fabricante} {self.modelo}'
    return nome_longo.title()

  def mostrar_km(self):
    # Exibe uma frase que mostrará a kilometragem do carro.
    print(f'O carro tem {self.kilometragem} KM rodados!')


novo_carro = Carro('audi', 'a4', 2016)
print(novo_carro.descricao())
novo_carro.mostrar_km()
```

```
2016 Audi A4
O carro tem 0 KM rodados!
```

13.11.1. Modificar valores em instâncias

Como vimos no exemplo 3, criamos uma instância, chamada ‘kilometragem’ que recebe um valor default igual a 0. Esse valor também pode ser modificado para alguma coisa que o usuário desejar, ou que o próprio programador desejar usar.

Como um método funciona quase igual a uma função, o método input também funciona normalmente. Vamos pegar um novo exemplo e modificar para vermos as duas maneiras de atribuir valores para as instâncias

```python
# Exemplo para tópico 13.11.1

class Cliente:
  def __init__(self, nome, idade, mesa):
    self.nome = nome
    self.idade = idade
    self.mesa = mesa
    self.pedido = 'Nada'

  def pedidos(self):
    print(f'O cliente {self.nome.title()}, da mesa {self.mesa.upper()} ', end='')
    print(f'fez o pedido de {self.pedido.title()}!')
```

```
# DEFININDO OS VALORES PARA AS INSTÂNCIAS DIRETAMENTE!
cliente1 = Cliente('José', 54, 'A5')
cliente1.pedido = 'Pizza Tradicional'
# Modificando o valor default da instância!
print(cliente1.pedidos())

print('=' * 50)
# O USUÁRIO QUE IRÁ DEFINIR VALORES PARA INSTÂNCIAS!
nome = input('Qual o nome? ')
idade = int(input('Qual a idade? '))
mesa = input('Qual a mesa? ')
cliente2 = Cliente(nome, idade, mesa)
cliente2.pedido = input('Qual o pedido? ')
print(cliente2.pedidos())
```

```
O cliente José, da mesa A5 fez o pedido de Pizza Tradicional!
None
==================================================
Qual o nome? kaio
Qual a idade? 43
Qual a mesa? m8
Qual o pedido? nenhum
O cliente Kaio, da mesa M8 fez o pedido de Nenhum!
```

13.11.2. Modificando o valor de um atributo com um método

Pode ser conveniente ter métodos que atualizem determinados atributos para você. Em vez de acessar o atributo de modo direto, passe o novo valor para um método que trate a atualização internamente.

Vamos continuar utilizando o exemplo da Classe Carro (EX3):

```python
# Exemplo para tópico 13.11.2

class Carro:
  # Tentativa simples de representar um carro!
  def __init__(self, fabricante, modelo, ano):
    # Vai inicializar os atributos que descrevem um carro!
    self.fabricante = fabricante
    self.modelo = modelo
    self.ano = ano
    self.kilometragem = 0  # instância para ler os KM (odometer_reading)
    # Essa instância também tem um valor default!

  def descricao(self):  #get_descriptive_name
    # Devolve um nome descritivo, formatado de modo elegante.
    nome_longo = f'{str(self.ano)} {self.fabricante} {self.modelo}'
    return nome_longo.title()

  def mostrar_km(self): #read_odometer
    # Exibe'' uma frase que mostrará a kilometragem do carro.
    print(f'O carro tem {self.kilometragem} KM rodados!')

  def atualizar_km(self, new_km):
    """Define o valor de leitura do hodômetro com o valor especificado.
    Rejeita a alteração se for tentativa de definir um valor menor para o
    hodômetro"""
    if new_km >= self.kilometragem:
      self.kilometragem = new_km
    else:
      print('Não é possível diminuir o valor lido na kilometragem!')

novo_carro = Carro('audi', 'a4', 2016)
print(novo_carro.descricao())
novo_carro.atualizar_km(50)
novo_carro.mostrar_km()
```

```
2016 Audi A4
O carro tem 50 KM rodados!
```

Explicação:

Agora, atualizar_km() verifica se o novo valor da kilometragem faz sentido antes de modificar o atributo.

Se a nova milhagem for maior ou igual à milhagem existente, self.kilometragem, você poderá atualizar o valor de leitura da kilometragem com a nova milhagem.

Se a nova milhagem for menor que a milhagem existente, você receberá um aviso informando que não pode diminuir o valor lido na kilometragem.

13.11.3. Aprendendo a usar o método __init__ de uma classe filha

Vamos continuar nosso aprendizado utilizando os exemplos de carros. Dessa vez, vamos modelar um carro elétrico, e ele é apenas um tipo específico de carro, portanto podemos basear nossa nova classe **CarroEletrico** na classe **Carro** que escrevemos antes.

Vale relembrar também que, quando formos criar uma classe-filha (subclasse) temos que especificar entre parênteses na definição da classe-filha.

```python
# Exemplo para tópico 13.11.3 - Sobre Herança e __init__() da subclasse

class Carro:
  # Tentativa simples de representar um carro!
  def __init__(self, fabricante, modelo, ano):
    # Vai inicializar os atributos que descrevem um carro!
    self.fabricante = fabricante
    self.modelo = modelo
    self.ano = ano
    self.kilometragem = 0  # instância para ler os KM (odometer_reading)
    # Essa instância também tem um valor default!

  def descricao(self):  #get_descriptive_name
    # Devolve um nome descritivo, formatado de modo elegante.
    nome_longo = f'{str(self.ano)} {self.fabricante} {self.modelo}'
    return nome_longo.title()

  def mostrar_km(self): #read_odometer
    # Exibe'' uma frase que mostrará a kilometragem do carro.
    print(f'O carro tem {self.kilometragem} KM rodados!')
```

```python
  def atualizar_km(self, new_km):
    """Define o valor de leitura do hodômetro com o valor especificado.
    Rejeita a alteração se for tentativa de definir um valor menor para o
    hodômetro"""
    if new_km >= self.kilometragem:
      self.kilometragem = new_km
    else:
      print('Não é possível diminuir o valor lido na kilometragem!')

class CarroEletrico(Carro):
  # Representa aspectos específicos de veículos elétricos.
  def __init__(self, fabricante, modelo, ano):
    # Inicializa os atributos da classe-pai.
    super().__init__(fabricante, modelo, ano)

meu_tesla = CarroEletrico('tesla', 'modelo s', 2016)
print(meu_tesla.descricao())
```

```
2016 Tesla Modelo S
```

→ As partes iniciais são a mesma de antes. Quando criamos uma classe-filha, a classe-pai deve fazer parte do arquivo atual e deve aparecer antes da classe-filha no arquivo.

→ O método __init__() aceita as informações necessárias para criar uma instância de Carro.

→ A função **super()** é uma função especial que ajuda Python a criar conexões entre a classe-pai e a classe-filha. Essa linha diz a python para chamar o método __init__() da classe-pai de CarroEletrico, que confere todos os atributos da classe-pai a CarroEletrico.
→ O nome **super** é derivado de uma convenção segundo a qual a classe-pai se chama (superclasse) e a classe-filha é a subclasse.

13.11.4. Definindo atributos e métodos da classe-filha

Depois que tiver uma classe-filha que herde de uma classe-pai, você pode adicionar qualquer atributo ou método novos necessários para diferenciar a classe-filha da classe-pai.

Vamos acrescentar um atributo que seja específico aos carros elétricos (uma bateria, por exemplo) e um método para mostrar esse atributo. Armazenaremos a capacidade da bateria e escreveremos um método que mostre uma descrição dela.

Obs.: Não vai ser colocado todo o código aqui novamente, apenas algumas partes e as partes novas.

```python
# Exemplo para tópico 13.11.4 - Sobre definição de atributos da classe-filha

class Carro:
  ...
class CarroEletrico(Carro):
  # Representa aspectos específicos de veículos elétricos.
  def __init__(self, fabricante, modelo, ano):
    # Inicializa os atributos da classe-pai.
    super().__init__(fabricante, modelo, ano)
    self.porc_bateria = 70

  def descricao_bateria(self):
    # Exibe uma frase que descreve a capacidade da bateria.
    print(f'O carro está atualmente com {str(self.porc_bateria)} - kWh.')

meu_tesla = CarroEletrico('tesla', 'modelo s', 2016)
print(meu_tesla.descricao())
meu_tesla.descricao_bateria()
```

```
2016 Tesla Modelo S
O carro está atualmente com 70 - kWh.
```

14. Arquivos

Aprender a trabalhar com arquivos e a salvar dados deixará seus programas mais fáceis de usar. Os usuários poderão escolher quais dados devem fornecer e quando. Aprender a tratar exceções também ajudará a lidar com situações em que os arquivos não existam e com outros problemas que possam fazer seus programas falharem.

Um arquivo de computador é um recurso de armazenamento de dados que está disponível em todo tipo de dispositivo computacional, seja um computador, dispositivo móvel, uma câmera fotográfica, entre outros.

Nota: verificar seção 7 do livro Python3 – Conceitos e aplicações, para melhor explanação do que é arquivo e seus tipos, além de revisar sobre arquivo binário que será importante na área de dados.

14.1. Ler dados de um arquivo

Ler dados de um arquivo é particularmente útil em aplicações de análise de dados, mas também se aplica a qualquer situação em que você queira analisar ou modificar informações armazenadas em um arquivo. Quando quiser trabalhar com as informações de um arquivo-texto, o primeiro passo será ler o arquivo em memória. Você pode ler todo o conteúdo de um arquivo ou pode trabalhar com uma linha de cada vez.

14.1.1. Lendo um arquivo inteiro

Vamos começar criando um arquivo simples, contendo algumas linhas de texto. Esse arquivo será

chamado ‘*duct.txt*’. Dentro desse arquivo está contido alguns nomes de pessoas:

```
José
Maria
márcia
Felipe
Jeniffer
Luiza
Pedro
john
faraday
```

Agora vamos fazer um programa que nós permitirá fazer a leitura de todo o arquivo de uma única vez.

```
caminho_arquivo = 'C:\\Users\\CVC\\Área de Trabalho\\Python\\exempl\\duct.txt'

with open(caminho_arquivo) as arquivo:
    leitura = arquivo.read()
    print(leitura)
```

Explicação:

A variável ‘*caminho_arquivo*’ é puramente para armazenar a localização de onde está o arquivo, caso o arquivo estivesse na mesma localização do programa (leitor.py) não precisaria pegar todo o caminho e poderia simplesmente colocar o nome do arquivo (como vamos ver mais a frente).

A palavra reservada **with** fecha o arquivo depois que não for mais necessário acessá-lo. Dá para trocar por close(), mas se um bug em seu programa impedir que a instrução close() seja executada, o arquivo não será fechado.

Para realizar qualquer tarefa com um arquivo, mesmo que seja apenas exibir seu conteúdo, você precisará inicialmente *abrir* o arquivo para acessá-lo. A função **open()** é justamente isso que faz, abre o arquivo. Ela precisa de um argumento dentro dos parênteses: o nome do arquivo que você quer abrir.
E no caso, pode ser passado tanto como uma variável que recebe o caminho (como o exemplo), ou pode ser passado diretamente dentro dos parênteses.

O método read() é basicamente para ler todo o conteúdo do arquivo e armazená-lo em uma longa string em ‘*leitura*’.

14.1.2. Lendo dados linha a linha

Quando estiver lendo um arquivo, com frequência você vai querer analisar cada linha do arquivo. Talvez você esteja procurando determinada informação no arquivo ou queira modificar o texto do arquivo de alguma maneira.

Para isso, podemos usar um laço for no arquivo para analisar cada uma de suas linhas,

```
JosÃ©
maria
marcia
felipe
jeniffer
luiza
pedro
john
faraday
```

uma de cada vez.

```
caminho_arquivo = 'C:\\Users\\CVC\\OneDrive\\Área de Trabalho\\Python\\exempl\\duct.txt'

with open(caminho_arquivo) as arquivo:
    for linha in arquivo:
        print(linha)
```

A saída do nosso programa vai resultar com algumas linhas em branco que não contém no arquivo original. Essas linhas em branco aparecem porque um caractere invisível de quebra de linha está no final de cada linha do arquivo-texto. A instrução print adiciona a sua própria quebra de linha sempre que a chamamos, portanto acabamos com dois caracteres de quebra de linha no final de cada linha: um do arquivo e outro da instrução print. Se usarmos rstrip() em cada linha na instrução print, eliminamos essas linhas em brancos extras.

14.1.3. Criando uma lista de linhas de um arquivo

Quando usamos o **with**, o objeto arquivo devolvido por open() (que é a variável) estará disponível somente no bloco with que o contém. Se quiser preservar o acesso ao conteúdo de um arquivo fora do bloco with, você pode armazenar as linhas do arquivo em uma lista dentro do bloco e então trabalhar com essa lista.

Pode processar partes do arquivo imediatamente e postergar parte do processamento de modo que seja feito mais tarde no programa.

```
caminho_arquivo = 'C:\\Users\\CVC\\OneDrive\\Área de Trabalho\\Python\\exempl\\duct.txt'

with open(caminho_arquivo) as arquivo:
    linhas = arquivo.readlines()
    for linha in linhas:
        print(linha.rstrip())
```

O método **readlines()** armazena cada linha do arquivo em uma lista. Essa lista é então armazenada em **linhas**, com a qual podemos continuar trabalhando depois que o bloco with terminar.

Usamos um laço for simples para exibir cada linha de **linhas**. Como cada item de **linhas** corresponde a uma linha do arquivo, a saída será exatamente igual ao conteúdo do arquivo.

14.2. Caminho de arquivo no Windows

Quando um nome de arquivo simples é passado para a função open(), Python observa o diretório em que o arquivo executado no momento está armazenado. Mas, às vezes, o arquivo que você quer abrir não estará no mesmo diretório que o seu arquivo de programa.

Para fazer Python abrir arquivos de um diretório que não seja aquele em que seu arquivo de programa está armazenado, é preciso fornecer um path de arquivo (caminho), que diz a python para procurar em um local específico de seu sistema.

No linux e no mac as coisas são fáceis, apenas copiar e colar o caminho já resolve. Mas no Windows não é a mesma coisa. Em sistemas Windows, use duas barras invertidas (\\) para separar as pastas.

Explicando no exemplo é mais fácil de entender:

O arquivo atual (test.py) está na pasta cap10

```
# abrindo arquivo no mesmo diretório que o programa
with open('frase.txt') as a:
    leitura = a.read()
    print(leitura)
```

```
ESSA CASA E DE ARRASAR
```

```
# abrindo arquivo em outro diretório
with open('C:\\Users\\CVC\\Área de Trabalho\\Python\\frase2.txt') as b:
    leitura2 = b.read()
    print(leitura2)
```

```
>>> ESSA CASA E DE ARRASAR <<<
MAS O ARQUIVO ESTA
EM OUTRO DIRETORIO
```

14.3. Modos de manipulação de arquivo

Como já vimos, para abrir um arquivo com Python é necessário a utilização da função *open()*. E vimos também que a função precisa de parâmetros, o parâmetro obrigatório que já vimos é basicamente o nome do arquivo.

Mas também existem outros parâmetros para se colocar entre os parênteses. Vamos olha para a sintaxe: *open('nome', 'modo', 'buffering')*. O parâmetro 'modo' é a forma de abertura do arquivo, que é o que iremos ver aqui na seção. O buffering é uma coisa que não precisamos nos preocupar muito no início, visto que esse parâmetro é a quantidade de bytes reservados na memória para a abertura do arquivo, e ele pode ser omitido, assim como o modo.

Os modos podem variar, sendo só leitura, só escrita ou os dois juntos. Os valores mais comumente utilizados para parâmetro 'modo' são:

r	– Somente para leitura
w	– Escrita (*caso o arquivo já exista, ele será apagado e um novo arquivo será criado*)
x	– Criação (*Permite a criação do arquivo exclusivamente se este não existir. Caso exista, levanta a exceção 'FileExistsError'*
a	– Leitura e Escrita ao final (*adiciona o novo conteúdo ao fim do arquivo*) (é como um append()
b	– Modo binário
t	– Modo Texto
+	– Leitura e Escrita (*permite adicionar leitura e escrita no arquivo. Independente de qual*

	seja o modo.
	– r+ → leitura e escrita
	– w+ → Escrita (o modo w+, assim como o w, apaga o conteúdo anterior do arquivo)
	– a+ → Leitura e escrita (*a diferença é que ele abre o arquivo p/ atualização*)

14.4. Método úteis para arquivos

Para os exemplos dos métodos, utilizaremos um arquivo chamado 'arquivoteste.txt' que conterá, inicialmente, uma frase. E ao decorrer do método vamos mudando.

'Olá Mundo!'
'Vamos continuar observando!'

14.4.1. Método write (e writelines)

Às vezes, você pode querer excluir um conteúdo do arquivo e substituí-lo totalmente por um novo conteúdo. Podemos fazer isso utilizando o método write().

Escrever no arquivo e apagar o anterior:

```
# Escrever no arquivo e apagar o anterior
with open(caminho_arquivo, 'w') as arquivo:
    arquivo.write('O conteudo anterior foi apagado')
```

```
O conteudo anterior foi apagado
```

Escrever no arquivo e deixar o conteúdo anterior:

```
# Escrever no arquivo e permanece o conteúdo anterior
with open(caminho_arquivo, 'a+') as arquivo:
    arquivo.write('\nNOVA INFORMACAO PARA O ARQUIVO')
```

```
O conteudo anterior foi apagado
================================
NOVA INFORMACAO PARA O ARQUIVO
```

O método **writelines()** vai servir para escrever várias linhas de uma única vez.

```
# Escrever no arquivo e permanece o conteúdo anterior
with open(caminho_arquivo, 'a+') as arquivo:
    arquivo.writelines(['\nMaria\n', 'Jefferson\n'])
```

```
O conteudo anterior foi apagado
================================
NOVA INFORMACAO PARA O ARQUIVO
Maria
Jefferson
```

14.4.2. Método read, readline e readlines

Os métodos para ler um arquivo são simples e que já vimos nos tópicos anteriores. Para lermos um arquivo, precisamos ter uma forma de 'interagir' com ele em nosso programa e é exatamente isso que métodos fazem.

14.4.2.1. Read()

O primeiro método que precisamos aprender retorna todo o conteúdo do arquivo com uma string.

```
with open(caminho_arquivo) as arquivo:
    print(arquivo.read())
```

```
Ola Mundo!
Vamos continuar observando!
```

14.4.2.2. Readline() x Readlines()

Podemos ler o arquivo linha por linha através desses dois métodos. Mas eles têm uma pequena diferença.

Readline() lê uma linha do arquivo até o fim dessa linha.
É possível, como opção, passar o tamanho, o número máximo de caracteres que desejamos incluir no resultado da string.

Já **Readlines()** retorna uma lista com todas as linhas do arquivo como elementos individuais (strings).

```
caminho_arquivo = 'arquivoteste.txt'

# Método readline devolvendo apenas uma linha
with open(caminho_arquivo) as arquivo:
    print(arquivo.readline())

print('#=' * 20, '\n')

# Método readlines devolvendo uma lista
with open(caminho_arquivo) as arquivo:
    print(arquivo.readlines())
```

```
Ola Mundo!

#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=

['Ola Mundo!\n', 'Vamos continuar observando!']
```

14.4.3. Método seek

A função seek() é usada para alterar a posição do identificador de arquivo (é como um ‘cursor’) para uma determinada posição. O identificador de arquivo define de onde os dados devem ser lidos ou gravados no arquivo.

Então imagine um arquivo com 4 linhas, o identificador estará no final dela, se eu mandar ler, ele vai começar a ler do final, por que o identificador estará no fim.

```
with open(caminho_arquivo, 'w+', encoding='utf8') as arquivo:
    arquivo.writelines((
        'Linha1\n', 'Linha2\n', 'Linha3\n', 'Linha4\n',
    ))
    print(arquivo.read())
```

Agora utilizaremos a função **seek()** para determinar onde ficará o identificador, que no caso será no inicio da linha:

```
with open(caminho_arquivo, 'w+', encoding='utf8') as arquivo:
    arquivo.writelines((
        'Linha1\n', 'Linha2\n', 'Linha3\n', 'Linha4\n',
    ))
    arquivo.seek(0, 0)
    print(arquivo.read())
```

```
Linha1
Linha2
Linha3
Linha4
```

14.4.4. Excluir arquivos

Para excluir arquivos utilizando Python, é preciso importar um módulo chamado **os**, que contém as funções para interagir com o seu sistema operacional.

Dentro do módulo existe a função remove() ou unlink() que recebe o caminho do arquivo como argumento e o exclui automaticamente.

```
import os

caminho2 = 'arquivo2.txt'
with open(caminho2, encoding='utf8') as arquivo:
    print(arquivo.read())

os.remove(caminho2)  # ou os.unlink(caminho2) que também funciona!
```

14.4.5. Renomear arquivos

Dentro do módulo **os** também é possível utilizar uma função para renomear ou mover o arquivo. E essa função é o **rename()**.

```python
import os

caminho2 = 'arquivo2.txt'
with open(caminho2, 'w+', encoding='utf8') as arquivo:
    arquivo.write('O arquivo atual vai ser renomeado!')

os.rename(caminho2, 'arquivo2.2-renomeado.txt')
# primeiro passamos o nome antigo, depois o novo nome.
```

Como usamos uma variável para armazenar o local de onde está o arquivo, ou simplesmente o nome do arquivo, dá para movermos também utilizando o rename, que no caso, seria apenas colocar no novo nome, o local de onde quer mover.

14.4.6. Enconding

15. Exceções

Python usa objetos especiais chamados exceções para administrar erros que surgirem durante a execução de um programa. Sempre que ocorrer um erro que faça Python não ter certeza do que deve fazer em seguida, um objeto exceção será criado. Se você escrever um código que trate a exceção, o programa continuará executando.

Se a exceção não for tratada, o programa será interrompido e um traceback, que inclui uma informação sobre a exceção levantada, será exibido. As exceções são tratadas com blocos try-except. Um bloco try-except pede que Python faça algo, mas também lhe diz o que deve ser feito se uma exceção for levantada.

Ao usar blocos try-except, seus programas continuarão a executar, mesmo que algo comece a dar errado.

15.1. Try, except, eles e finally

- **try**: Bloco de código a ser executado;
- **except** {exceção}:
  - Código que será executado caso a exceção seja capturada;
  - A exceção pode ser omitida.
- **else**:
  - Código que será executado caso nenhuma exceção tenha sido lançada ou capturada;
- **finally**:
  - Código que será executado independente se alguma exceção for capturada ou não.

Apenas as cláusulas **try** e **except** são obrigatórias, sendo **else** e **finally** opcionais! Além disso, é possível ter múltiplas cláusulas **except**, capturando exceções diferentes.

Vamos observar um erro simples, e ele é A **exceção ZeroDivisionError:**

```
# Exemplo de erro de ZeroDivionError
a = 10
b = 0

# Se tentarmos rodar assim, ira apresentar um erro!
# print(a / b)

try:
    print(a / b)

except:
    print('Divisão Impossivel')
```

Vamos a outro **exemplo:**

```
try:
    print('ABRINDO A PORTA!')
except:
    print('UM ERRO OCORREU!')
    # se houvesse um erro em 'try', então
    # o bloco except seria executado
else:
    print('NENHUM ERRO OCORREU!')
finally:
    print('PORTA FECHADA!')
```

```
ABRINDO A PORTA!
NENHUM ERRO OCORREU!
PORTA FECHADA!
```

Python também disponibiliza uma lista com algumas das exceções, sendo uma das maiores a exceção 'Exeption' e as outras, como *ZeroDivionError*, uma sub-exceção.
Para acessar todas, basta ir no link: https://docs.python.org/pt-br/3/library/exceptions.html

15.2. Tratando exceções para arquivos

15.2.1. FileNotFoundError

Um problema comum ao trabalhar com arquivos é o tratamento de arquivos ausentes. O arquivo que você está procurando pode estar em outro lugar, o nome do arquivo pode estar escrito de forma incorreta ou o arquivo talvez simplesmente não exista. Podemos tratar todas essas situações de um modo simples com um bloco try-except.

```
1  caminho_arquivo = 'arquivo_excecoes.txt'
2
3  try:
4      with open(caminho_arquivo) as arquivo:
5          leitura = arquivo.read()
6          print(leitura)
7  except FileNotFoundError:
8      print('O arquivo não foi encontrado!')
9      print('Verifique se o diretorio está correto!')
```

– O arquivo não está criado!

```
O arquivo não foi encontrado!
Verifique se o diretorio está correto!
```

15.2.2. Criando e lendo um arquivo com exceções

Python → modulo 6 → exer115 → arquivo → __init__.py

15.3. **Armazenando Dados**

Haverá casos em que precisaremos armazenar tipos de dados, e as vezes nós teremos que pedir aos usuários que forneçam determinados tipos de informação.

Qualquer que seja o foco de seu programa, você armazenará as informações fornecidas pelos usuários em estruturas de dados como listas e dicionários.

Quando os usuários fecham um programa, quase sempre você vai querer salvar as informações que eles forneceram. Uma maneira simples de fazer isso envolve armazenar seus dados usando o módulo json.

O módulo json permite descarregar estruturas de dados Python simples em um arquivo e carregar os dados desse arquivo na próxima vez que o programa executar. Também podemos usar json para compartilhar dados entre diferentes programas Python.

NOTA: O formato JSON (JavaScript Object Notation, ou Notação de Objetos JavaScript) foi originalmente desenvolvido para JavaScript. Apesar disso, tornou-se um formato comum, usado por muitas linguagens, incluindo Python.

NOTA2: O formato JSON é muito simples, ela não suporta coisas que executam ações específicas como, por exemplo, funções, métodos, ou classes. Além de outras coisas.

15.3.1. Utilizando json.dump() e json.load()

Para o exemplo, criarei uma variável que armazenará algumas informações sobre uma pessoa. Dentro dessa variável vai conter dicionários, listas, valores inteiros e booleanos.

E nesse programa, ele armazena as informações e logo depois ele traz de volta.

```python
import json  # precisa importar a biblioteca para usar json

# variável com algumas informações
pessoa = {
    'nome': 'Kaio',
    'sobrenome': 'Felipe',
    'enderecos': [
        {'rua': 'None', 'numero': 32},
        {'rua': 'Severino borges', 'numero': 55},
    ],
    'altura': 1.7,
    'numeros_preferidos': (2, 4, 6, 8, 10),
    'dev': True,
    'nada': None,
}

arquivojson = 'arquivo_pessoa.json'

# Parte para armazenar arquivo!
with open(arquivojson, 'w') as armazenar:
    json.dump(pessoa, armazenar)
    print('Informações Armazenadas!')
    # Como o programa não tem saída, coloquei
    # esse print apenas para saber se tudo foi OK!
    print('=' * 20)

with open(arquivojson, 'r', encoding='utf8') as exportar:
    # r -> modo leitura / enconding vai servir para os acentos!
    pessoa_exportando = json.load(exportar)
    # o load é p/ carregar um arquivo, e entre () precisa do nome do arquivo
    print(pessoa_exportando)
```

```
Informações Armazenadas!
====================
{'nome': 'Kaio', 'sobrenome': 'Felipe', 'enderecos':
[{'rua': 'None', 'numero': 32}, {'rua': 'Severino
borges', 'numero': 55}], 'altura': 1.7,
'numeros_preferidos': [2, 4, 6, 8, 10], 'dev': True,
'nada': None}
```

15.3.2. Salvando um dado gerado por usuário

Salvar dados com json é conveniente quando trabalhamos com dados gerados pelo usuário porque, se você não salvar as informações de seus usuários de algum modo, elas serão perdidas quando o programa parar de executar. Vamos observar um exemplo em que pedimos aos usuários que forneçam seus nomes na primeira vez em que o programa executar e, então, o programa deverá lembrar esses nomes quando for executado novamente.

```python
import json

arquivo_usuario = 'usuario.json'

try:
    # Carrega o nome do usuário se foi armazenado anteriormente
    # Caso contrário, pede que o usuário forneça o nome e armazena essa informação.
    with open(arquivo_usuario) as arquivo:
        nome = json.load(arquivo)
except FileNotFoundError:
    nome = input('Qual seu nome? ')
    with open(arquivo_usuario, 'w') as arquivo:
        json.dump(nome, arquivo)
        print(f'O nome - {nome} - foi armazenado!')
else:
    print(f'Bem vindo de Volta, {nome}!')
```

Referências

- **Métodos comuns para strings→**
  - http://devfuria.com.br/python/strings/
  - https://blog.betrybe.com/python/python-split/
  - https://www.acervolima.com.br/2021/01/python-string-split.html
  - https://www.pythonprogressivo.net/2018/10/String-Maiuscula-Minuscula.html
  - http://ptcomputador.com/P/python-programming/93783.html
  - https://acervolima.com/string-capitalize-em-python/
  - https://acervolima.com/metodo-python-string-count-1/
  - https://pt.stackoverflow.com/questions/249532/como-manipular-strings-com-find
- **Alinhamento de F-Strings →**
  - https://www.hashtagtreinamentos.com/f-strings-em-python#:~:text=As%20f%2Dstrings%20v%C3%A3o%20servir,vari%C3%A1vel%20dentro%20de%20%7B%7D%20chaves.&text=D%C3%A1%20s%C3%B3%20uma%20olhada%20como,informa%C3%A7%C3%B5es%20dentro%20de%20um%20texto.
  - https://acervolima.com/alinhamento-de-string-em-f-string-python/
- **Fatiamento de Strings →**
  - https://pense-python.caravela.club/08-strings/04-fatiamento-de-strings.html
- **Utilizações em Listas →**
  - https://www.freecodecamp.org/portuguese/news/o-metodo-sort-para-arrays-e-listas-em-python-ordenacoes-ascendente-e-descendente-explicadas-e-com-exemplos/#:~:text=O%20m%C3%A9todo%20sort()%20permite,valor%20intermedi%C3%A1rio%20para%20cada%20elemento.
  -
- **Utilizações em Tuplas →**
- **Utilizações em Dicionários →**
  - https://pythonacademy.com.br/blog/dicts-ou-dicionarios-no-python#:~:text=chave3'%3A%200%7D-,Operador%20de%20%E2%80%9Cdesempacotamento%E2%80%9D,e%20em%20nossos%20queridos%20dicion%C3%A1rios.
- **Módulos em Python →**
  -
- **Cores em Python →**

-

- **Orientação a Objetos →**
  - https://www.youtube.com/watch?v=dG7LlYne2VA&list=PLucm8g_ezqNqj--UUSn16yfDp3xcZi40t&index=2
  - https://cursos.alura.com.br/forum/topico-uso-do-self-136352
  - https://panda.ime.usp.br/aulasPython/static/aulasPython/aula17.html

- **Arquivos e Exceções →**
  - https://diegomariano.com/manipulando-arquivos/#:~:text=Para%20abrir%20um%20arquivo%20de,abertura%20do%20arquivo%20(opcional).
  - https://www.freecodecamp.org/portuguese/news/como-escrever-em-um-arquivo-em-python-open-read-append-e-outras-funcoes-de-manipulacao-explicadas/
  -