Funções de strings

1. Funções de *strings*

É possível fazer várias operações com *strings* utilizando técnicas como concatenação ou converter em listas. Porém, certas operações são muito comuns e podem ser muito trabalhosas de fazer na mão. Por isso, temos diversas funções prontas para nos ajudar.

Note que como *strings* são imutáveis, nenhuma dessas funções irá alterar a *string* original. Elas sempre **retornarão** uma *string* nova com as modificações desejadas.

Vejamos algumas das mais usadas.

1.1. Maiúsculas e minúsculas

Temos algumas funções prontas para alterar a capitalização das letras. Uma delas é a função *upper*, que transforma todas as letras da *string* original em maiúsculas:

```
frase = 'vAmOs PrOgRaMaR'
maiuscula = frase.upper()
print(maiuscula) # resultado: 'VAMOS PROGRAMAR'
```

Analogamente, temos a função *lower* para transformar todas as letras em minúsculas:

```
frase = 'vAmOs PrOgRaMaR'
minuscula = frase.lower()
print(minuscula) # resultado: 'vamos programar'
```

Também é possível formatar a *string* inteira como um nome próprio: primeira letra de cada palavra maiúscula, todo o restante em minúscula. Para isso temos a função *title*:

```
frase = 'vAmOs PrOgRaMaR'
titulo = frase.title()
print(titulo) # resultado: 'Vamos Programar'
```

E, por fim, é possível tratar nossa *string* como uma frase gramaticalmente correta: primeira letra maiúscula, todo o resto minúsculo. Essa função é a *capitalize*:

```
frase = 'vAmOs PrOgRaMaR'
correta = frase.capitalize()
print(correta) # resultado: 'Vamos programar'
```

Uma utilidade para essas funções é padronizar entrada de usuário. Quando pedimos para o usuário digitar 'sim' caso ele deseje fazer algo, ele pode digitar 'SIM', 'sim', 'Sim', 'sIm', 'siM', 'SIm', 'sIM' ou 'SiM'. Prever todas essas condições em uma condicional pode ser bastante trabalhoso, ou mesmo impossível. Imagine se fosse uma *string* de várias letras... Porém, podemos forçar um padrão para a entrada do usuário e comparar com esse padrão:

```
usuario = input('Digite "sim" se aceita os termos de uso: ')
if usuario.upper() == 'SIM':
  print('Seja bem-vindo!')
```

```
else:
  print('Que pena.')
```

Qualquer forma que o usuário digite a palavra 'sim' será aceita.

1.2. Quebrando uma *string*

É possível separar uma *string* em uma lista de *substrings*. Isso pode ser particularmente útil quando precisamos separar um texto em palavras individuais. A função que realiza essa quebra é o *split*:

```
texto = 'uma frase qualquer'
palavras = texto.split()
print(palavras) # resultado: ['uma', 'frase', 'qualquer']
```

O *split* é mais do que apenas uma função para separar palavras. Podemos opcionalmente passar como parâmetro uma *string* para ser usada como critério de separação: ao invés de quebrar no espaço em branco, a *string* principal será quebrada nos pontos onde o parâmetro aparece (e ele será apagado do resultado).

Uma utilidade interessante para isso seria quando estamos interessados em ler dados formatados do teclado ou de um arquivo e pegar as informações que nos interessam. Imagine, por exemplo, que você queira que o usuário digite uma data no formato 'dd/mm/aaaa' e em seguida você precise separar dia, mês e ano em três variáveis do tipo **int**. Isso é possível com o *split*:

```
data = input('Digite uma data: ')
lista_data = data.split('/')
dia = int(lista_data[0])
mes = int(lista_data[1])
ano = int(lista_data[2])
print('Dia: ', dia)
print('Mês: ', mes)
print('Ano: ', ano)
```

1.3. Substituindo elementos na *string*

Uma das ferramentas mais úteis em qualquer editor de texto é o *localizar e substituir*, onde podemos buscar por todas as ocorrências de uma expressão no texto e trocar por outra expressão. Em Python temos uma função análoga, o *replace*. Ele recebe 2 parâmetros: a expressão a ser substituída e a expressão que a substituirá. Veja o exemplo:

```
frase = 'Python é difícil. Por ser difícil, devemos estudar.'
corrigida = frase.replace('difícil', 'fácil')
print(corrigida)
# resultado: 'Python é fácil. Por ser fácil, devemos estudar.'
```

Em Python não existe uma função para deletar um pedaço de uma *string*. Porém, podemos usar o replace para substituir uma expressão por uma *string* vazia, o que tem o mesmo resultado:

```
palavra = 'batata'
consoantes = palavra.replace('a', '')
print(consoantes)
# resultado: 'btt'
```

1.4. Concatenando *strings* em uma coleção

Imagine que você tem uma coleção (por exemplo, uma lista) de *strings* e precisa unir todas elas utilizando algum símbolo padrão como separador entre elas. Para isso temos o *join*. Ele soa pouco intuitivo no começo, então convém executar o exemplo e observar com atenção seu resultado:

```
lista = ['a', 'b', 'c']
separador = '123'
resultado = separador.join(lista)
print(resultado) # resultado: 'a123b123c'
```

Já vimos o *join* antes sendo usado para converter uma lista de volta em *string*. Para isso, utilizávamos uma *string* vazia como separador. Assim, os elementos da lista eram concatenados sem separador.

> **Dica:** um jeito de memorizar facilmente como o *join* funciona é pensar que o separador entrará no lugar das vírgulas na visualização da lista.

2.0. Formatando *strings*

2.1. A função *format*

Todo mundo que já preencheu um contrato ou uma ficha de cadastro está familiar com textos nesse estilo:

> Eu, _____, portador do CPF _____, residente no endereço _____ autorizo o procedimento.

Esse é o texto genérico que vale para todos, e cada um de nós em particular entende que deve preencher os campos em brancos com dados específicos (nome, CPF e endereço, no exemplo acima).

Existe uma função em Python para realizar esse tipo de preenchimento de texto: o *format*. Suponha que você tenha dados em diferentes variáveis e precisa que todos eles apareçam em uma *string*. Basta criar uma *string* com os "espaços em branco" para serem preenchidos e passar as variáveis para a função. Os espaços em branco são representados por chaves ({}).

```
nome = input('Digite seu nome: ')
cpf = input('Digite seu cpf: ')
endereco = input('Digite seu endereço: ')
contrato = 'Eu, {}, portador do CPF {}, residente no endereço {} autorizo o procedimento.'
contrato_preenchido = contrato.format(nome, cpf, endereco)
print(contrato_preenchido)
```

Porém, o grande charme não está apenas em preencher - isso poderia ser feito concatenando com o operador +. Nós podemos colocar opções de formatação nos nossos dados, como número de casas em um número.

Imagine que você queira exibir uma data no formato dd/mm/aaaa. Em situações normais, dias e meses inferiores a 10 apareceriam com apenas 1 casa (**int** não é representado com zeros à esquerda). Porém, podemos especificar no *format* que gostaríamos de representar um inteiro com 2 casas, preenchendo com zero casas em branco.

```
dia = 1
mes = 2
ano = 2020
data = '{:02d}/{:02d}/{:04d}'.format(dia, mes, ano)
print(data) # resultado: 01/02/2020
```

Vamos entender o que está dentro das chaves: o símbolo **:** indica que passaremos opções. O símbolo 'd' indica que estamos representando números inteiros em base decimal (dígitos de 0 a 9). Os símbolos '2' e '4' indicam, respectivamente, 2 casas ou 4 casas. E o símbolo '0' indica que se faltar dígitos, os espaços devem ser preenchidos com zero.

Vejamos outro exemplo, dessa vez com casas decimais. É normal postos de gasolina mostrarem o preço do litro com 3 casas decimais. Mas o preço final a ser cobrado deverá ter 2 casas. Porém, ao multiplicar o preço por litro pelo valor em litros, é provável que o total dê várias casas decimais. Usaremos o *format* para representar com apenas 2 casas.

```
preco_litro = 5.234
litros = 29.5
total = preco_litro * litros
print(total) # resultado: 154.403
```

```
preco_final = 'R$ {:.2f}'.format(total)
print(preco_final) # resultado: R$ 154.40
```

Neste caso, o 'f' indica que o número é **float**. Já o '.2' indica que queremos 2 casas após o ponto decimal. Note que ele não apenas descarta as casas excedentes, e sim arredonda corretamente o número.

> O *format* possui tantas opções diferentes que existe um site inteiro dedicado a explicar e dar exemplos: https://pyformat.info/

2.2. *f-strings*

A função *format* já se mostrou bastante poderosa, e vimos como ela facilita muito nossa vida. Ela não apenas diminui nosso trabalho para concatenar múltiplas informações em uma única string, como ainda aceita opções diversas de formatação para cada uma das informações. Mas dá para ir um passo além.

Ao colocarmos o caractere `f` imediatamente antes de abrir aspas para criar uma string, essa string irá se comportar de maneira muito semelhante à máscara de um *format*, e não será necessário chamar o *format*. Podemos colocar o nome de variáveis entre chaves para inseri-las automaticamente na string:

```
nome = input('Digite seu nome: ')
cpf = input('Digite seu cpf: ')
endereco = input('Digite seu endereço: ')
contrato = f'Eu, {nome}, portador do CPF {cpf}, residente no endereço {endereco} autorizo o procedimento.'
print(contrato)
```

Inclusive podemos utilizar as mesmas opções de formatação do *format* diretamente nas *f-strings*:

```
preco_litro = 5.234
litros = 29.5
total = preco_litro * litros
print(total) # resultado: 154.403

preco_final = f'R$ {total:.2f}'
print(preco_final) # resultado: R$ 154.40
```

Desde o seu lançamento, as *f-strings* vem se tornando cada vez mais populares, tornando o uso do *format* menos comum.