**String em Python**

**Extraindo informações de uma URL**

1. **Aula 1 – Introdução e Fatiamento de Strings:**
   1. Em uma URL existem parâmetros no final que funcionam como variáveis passadas para o nosso programa.
      1. Temos como exemplo essa imagem:



* + 1. Começa com https:// e o site em que estamos, em seguida, temos /Search? Que indica que estamos na página de pesquisa.
    2. O *q=* é a nossa primeira variável que indica query, ou seja, indica qual a busca/pesquisa que eu fiz. Nesse caso indicando alura.
    3. &: é o que faz a separação das variáveis.
    4. Src: source = indica origem, que nesse caso é typed\_query, ou seja, pesquisa digitada.
    5. Isso tudo indica para o twitter que fizemos uma pesquisa digitada por alura na página de busca do site twitter.
  1. Criando um exemplo: Precisamos fazer o cambio de moedas no site bytebank. O URL ficaria mais ou menos assim:
     1. <https://bytebank.com/cambio?moedaOrigem=real&moedaDestino&quantidade=100>
     2. Na URL acima temos que: queremos fazer o cambio de 100 unidades de dinheiro do real para dólar no site bytebank.
  2. Podemos fazer o fatiamento de strings.
     1. Quando setamos uma string em uma variável, podemos pedir para o python imprimir o que está em alguma posição, como na posição 0, por exemplo.
     2. O que python irá exibir é a letra que se encontra naquela posição. E.:



* + 1. Lembrando que a contagem de posições do python é de 0 até o -1 o len() da string.
    2. Também podemos pedir para que ele devolva o que está em mais de uma posição de uma vez, ou seja, conseguimos ver o que está da posição 1 desejada até a posição 2, separada por “:”. Ex.:



* + 1. Porém, no fatiamento, temos que a posição/argumento 1 é inclusivo e o 2 é exclusivo, ou seja, quando pedimos para exibir o que está da posição 0 até a 1, o que ele exibirá será somente a primeira letra, pois excluiu o que está na posição 1. Exs.:



* + 1. Porém, apesar de a segunda posição escolhida é exclusiva, podemos acrescentar um somador na hora de pedir a devolução do valor, desse modo tornando a segunda posição inclusiva também. Ex.:



* 1. Strings uma vez que colocadas se tornam imutáveis. Tudo o que fazemos com elas depois disso é apenas cópia da original.
     1. Para saber mais sobre imutabilidade: <https://blog.saldanha.dev/imutabilidade-de-strings-no-python>
  2. O que aprendemos:
     1. URLs e seus formatos: como as URLs funcionam e o que cada parte de uma URL significa - base e parâmetros;
     2. O operador de fatiamento [a:b], utilizado para obter uma substring desde o índice a até o índice b - 1 da string original. Lembrando que b - 1 pois o segundo argumento do fatiamento é exclusivo;
     3. A string original não é alterada ao ser fatiada devido à sua imutabilidade.

1. **Aula 2 – Utilizando Métodos de Strings:**
   1. O problema de usar o fatiamento para separar o que é site/domínio do que é parâmetro é que estamos considerando que em todas as url’s o “?” que faz essa separação está na mesma posição, o que não é verdade.
      1. Levando isso em consideração, o que nós temos que fazer é utilizar o método *find()*. Ex.:



* + 1. Repare que ele retornou 2, sendo essa a posição da letra na string. Lembrando que a contagem sempre começa em 0.
    2. Utilizando esse método, podemos fazer com que a separação sempre seja feita através da posição do “?” independentemente de onde seja. Ex.:



* + 1. No url\_parametros nós colocamos +1 justamente para retirar o “?” do fatiamento. Lembrando que a primeira posição é inclusiva e a segunda exclusiva, por isso na url\_base não precisamos colocar -1, por exemplo. Essa matemática já é feita por padrão.
  1. Um outro problema que temos é que o nsso final do parâmetros ainda está fixo, e nós não sabemos qual é o tamanho de todas as URL’s e ficar contando e alterando o nosso programa para cada uma delas não é muito prático.
     1. A nossa solução para isso é simples: Só deixar o segundo argumento de posição vazio, dessa forma ele irá pegar todo o resto que estiver na URL. Ex.:



* + 1. Dessa forma o python entende que você quer ir daquele início passado até o final da string.
    2. O mesmo vale para o início das strings, estávamos colocando o 0 até agora, mas podemos simplificar e deixar sem nada que o resultado será o mesmo. Ex.:

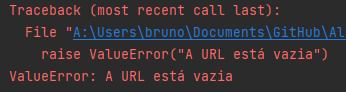


* 1. Documentação oficial do método find: <https://docs.python.org/pt-br/3/library/stdtypes.html?highlight=find#str.find>
  2. Com o leng() podemos verificar o tamanho de uma string e fazer com que a partir disso o find nos devolva o que vem depois dela.
  3. O que aprendemos:
     1. Uma string é uma cadeia de caracteres onde cada caractere tem sua própria posição ou índice;
     2. Podemos omitir o primeiro ou o segundo argumento do operador de fatiamento para fatiar uma string do início até um certo índice, ou a partir de um índice até o final. Exemplo: str[a:] ou str[:b];
     3. Podemos utilizar o método str.find(palavra, inicio) para buscar o índice de palavra a partir de início;
        1. Caso palavra não seja encontrada, o método find retorna -1.
     4. O método len(string) retorna o tamanho (ou seja, a quantidade de caracteres) da nossa string;
        1. Dica: o caractere que representa espaço em branco (“ “) também conta! Por exemplo: len(" ") retorna 1.

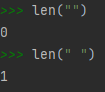
1. **Aula 3 – Aplicando Orientação a Objeto no Projeto:**
   1. Quando queremos levantar um erro caso o usuário faça algo que não deveria, podemos criar uma condição e utilizar o *raise ValueError()*, como se fosse um return.
      1. Isso indica para o python que houve um erro de valor podendo até mesmo mandar uma mensagem:



* + 1. A mensagem que aparece no console fica assim:



* + 1. Porém isso só vai acontecer quando estiver uma string vazia, sem espaços nem nada, mas, caso haja um espaço, ele vai rodar o programa e não vai exibir nada.
       1. Isso acontece porque a len() de uma string “” é igual a 0, mas a len() de uma string “ “ é igual a 1:



* + - 1. Portanto, “” == “ “ é false:

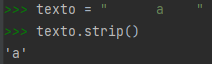


* + 1. Para concertar esse problema temos que fazer a sanitização dos dados, ou seja, limpar ele. Podemos criar uma sessão só para isso.
    2. A sanitização da remoção de espaços em branco pode ser feita de 2 maneira:
       1. .replace(o que será substituí, pela substituição): Substituí o primeiro argumento pelo segundo passado. Ex.:



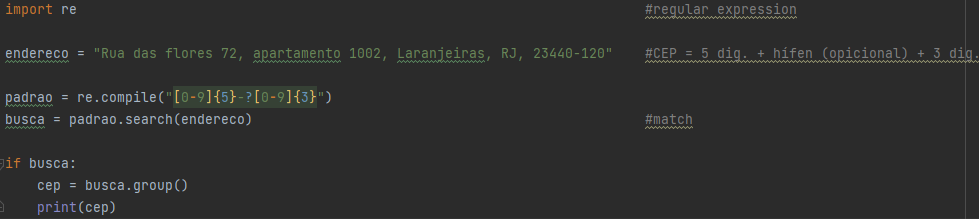


* + - 1. .strip(): Retira automaticamente todos os espaços em branco antes e depois de algo escrito

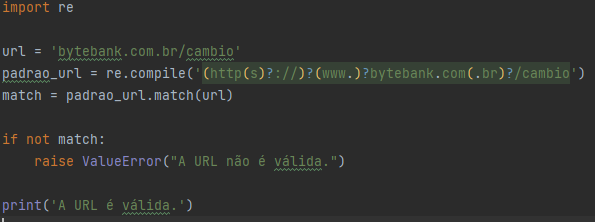


* + 1. Existem também variações do strip(), sendo:
       1. Lstrip(): Remove os espaços em branco da esquerda.
       2. Rstrip(): Remove os espaços em branco da direita.
    2. A principal diferença entre eles é que o replace não remove caracteres especiais como \t e \n, já o strip, sim.
  1. Para criar uma classe temos que saber mais ou menos como queremos que seja a interface dela: o que ela fará, os métodos, prints, dentre outros.
  2. Para saber mais sobre métodos do str do python: <https://docs.python.org/pt-br/3/library/stdtypes.html#string-methods>
  3. O que aprendemos:
     1. Podemos utilizar a palavra-chave raise para lançar uma exceção no nosso programa, informando ao usuário qual erro ocorreu;
     2. Mais métodos de strings: str.replace e str.strip;
     3. Como transformar um código em uma classe com atributos e métodos;
     4. A diferença entre None, ””, 0, e como o if do Python funciona;
     5. O operador not.

1. **Aula 4 – Expressões Regulares:**
   1. Para trabalhar com expressões regulares/RegEx, podemos importar a biblioteca *re* que é integrada ao python.
   2. Compile(“regex”): função da biblioteca acima para passarmos a regex.
   3. Search(variavel): função para procurar dentro de um padrão e retorna o macth, caso tenha, ou none, caso não encontre o que procura.
   4. Group(): Função que retorna exatamente o que foi encontrado no padrão.
   5. Utilizando essas funções, podemos criar um if para verificar e imprimir a regex que queremos encontrar. Ex.:



* 1. Basicamente estamos importando a biblioteca regex, atribuindo o endereço a uma variável, dizendo para o python que o nosso padrão é a compilação (a partir da biblioteca) da regex passada e buscando o padrão regex dentro de endereço. Se a busca retornar true, atribua a busca agrupada na variável cep e imprima ele.
  2. Se tiver alguma dúvida sobre como escrever regex, revisitar as anotações do curso de regex.
  3. Match(): método que faz a procura do padrão já citado em um url/str. Ex.:



* 1. Para saber mais sobre RegEx:
     1. Expressões Regulares é um tópico muito extenso na área de computação e muito importante. Praticamente toda linguagem de programação possui um conjunto de ferramentas para trabalhar com Expressões Regulares.
     2. O mais interessante é que, apesar de serem linguagens diferentes, o modo de utilizar vai ser muito parecido com o que vimos: estabelecer um padrão (RegEx) e aplicar esse padrão a um texto, ora para extrair o valor que esteja de acordo com o padrão fornecido (search), ora para verificar se o texto está de acordo com o padrão (match).
     3. Se você se interessou pelo assunto e quer se aprofundar ainda mais, a documentação oficial do Python tem uma seção de HOWTO (“como fazer”) voltada especificamente em como trabalhar com expressões regulares. Ela pode ser acessada através do link: https://docs.python.org/pt-br/3/howto/regex.html#regex-howto
  2. O que aprendemos:
     1. Como construir e utilizar expressões regulares, ou RegEx utilizando o módulo re do Python;
     2. Qual a diferença entre search e match;
     3. O que são quantificadores e intervalos em RegEx;
     4. A diferença entre parênteses (..) colchetes [...] na construção de padrões;
     5. Como utilizar expressões regulares para fazer validações complexas.

1. **Aula 5 – Métodos especiais:**
   1. \_\_len\_\_(sefl): método especial para incrementar no nosso objeto que retorna o tamanho dele quando perguntamos ao python. Ex.:



* 1. Para saber mais, o método dir():
     1. Nós utilizamos o editor PyCharm para inspecionar os métodos da classe str. Mas também podemos verificar se uma classe possui um determinado método de outra forma: utilizando o método dir(objeto).
     2. O método dir(objeto) é um método embutido (built-in) que retorna uma lista de strings com todos os métodos e atributos do objeto passado.
     3. Por exemplo, se fizermos dir(str) temos o seguinte retorno (simplifiquei devido a grande quantidade de métodos e atributos da classe str):

*dir(str)*

*[..., ‘\_\_len\_\_’, ... , ‘strip’, … ‘]*

* + 1. Inclusive, se você souber o nome do método que está procurando, pode verificar se ele está nessa lista utilizando o operador in:

*‘\_\_len\_\_’ in dir(str) # Verifica se o método \_\_len\_\_ existe na classe str*

*True*

* 1. \_\_str\_\_(self): método chamado quando damos o print() em algum objeto, sempre retorna o que queremos. Ex.:



* 1. \_\_eq\_\_(self, Other): método onde dizemos para o python que é para ele comparar algo específico do objeto com o outro objeto, e não a localização de memória, assim como ele faria normalmente. Ex.:



* 1. Esse é o conceito de que dois objetos podem ser iguais, mas não idênticos, é diferente da utilização de “===” ou “is”, por exemplo, que fazem a comparação do id() de cada objeto, ou seja, sua localização/endereço de memória.
  2. Diferença entre o is e o ==: <https://www.alura.com.br/artigos/qual-a-diferenca-entre-e-is-no-python>, para saber mais.
  3. Desafio para mais tarde: <https://cursos.alura.com.br/course/string-python-extraindo-informacoes-url/task/91909>. Resolução: <https://github.com/alura-cursos/manipulacao-strings-alura/tree/desafio>.