# Programação Script

## Strings

**Aula 10**

Prof. Felipe A. Louza

# Roteiro


1. Strings em **Python**
2. Concatenação de Strings
3. Algumas operações
4. Comparação de Strings
5. Exemplos com Strings
6. Referências

# Roteiro


1. Strings em **Python**
2. Concatenação de Strings
3. Algumas operações
4. Comparação de Strings
5. Exemplos com Strings
6. Referências

# Strings em **Python**

Strings em **Python** representam uma lista de caracteres.

- Strings podem ser escritas entre aspas simples ‘ ou entre aspas duplas ”.

```
>>> s = "amora"
>>> s
'amora'
```

```
>>> type(a)
<class 'str'>
```

- A `string` vazia é representada como ‘’ ou “”.

```
>>> type('')
<class 'str'>
```

# Strings em **Python**

Podemos atribuir mais de uma linha de caracteres para uma mesma `string`:

- Usamos 3 aspas simples ‘ ou aspas duplas ”.

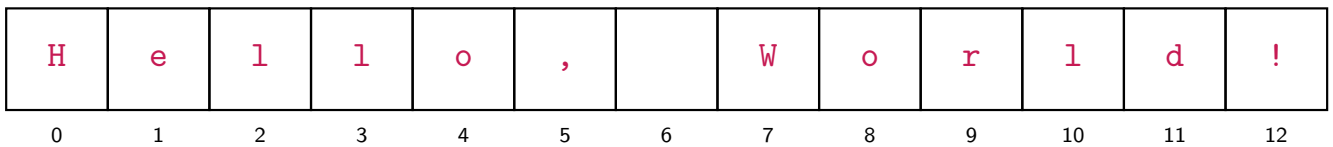
```
>>> s = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt."""
>>> s
'Lorem ipsum dolor sit amet,\nconsectetur adipiscing elit,\nsed do ...'
```

- O caractere `\n` só causa a mudança de linha no comando `print()`

```
>>> print(s)
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt.
```

# Strings em **Python**

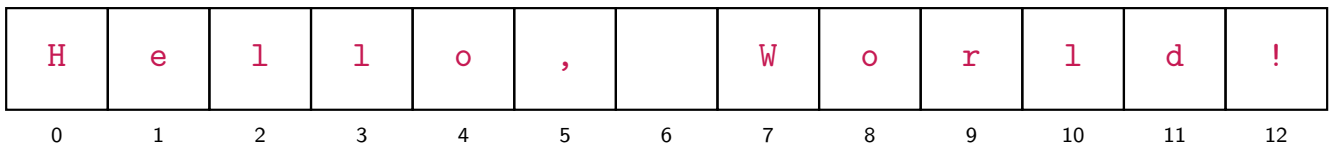
- Podemos **acessar** itens da lista, como vimos anteriormente.




```
1 >>> s = "Hello, World!"
2 >>> s[0]
3 'H'
4 >>> s[-1]
5 '!'
```

# Strings em Python

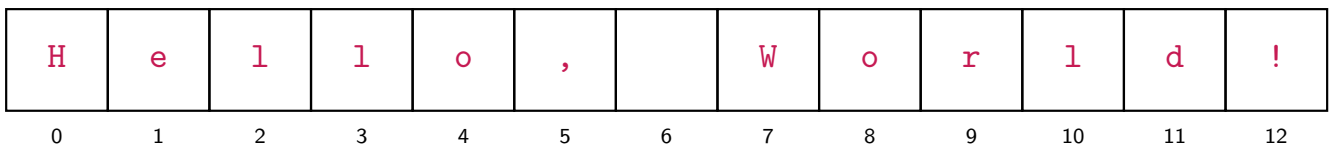
- Cuidado com posições inválidas:




```
1 >>> len(s)  #número de caracteres
2 13
3 >>> s[13]
4 Traceback (most recent call last):
5   File "<pyshell#17>", line 1, in <module>
6     s[13]
7 IndexError: string index out of range
```

# Strings em **Python**

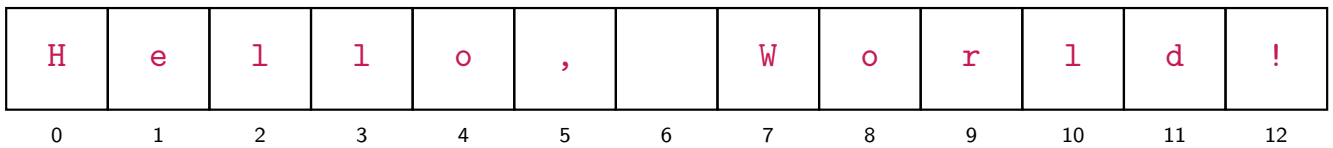
- Podemos fazer **slicing** com `strings`:




```
1 >>> s = "Hello, World!"
2 >>> s[2:5]
3 'llo'
4 >>> s[:5]
5 'Hello'
6 >>> s[7:]
7 'World!'
```

# Strings em Python

- Podemos percorrer todos os itens (caracteres) com um **laço**:

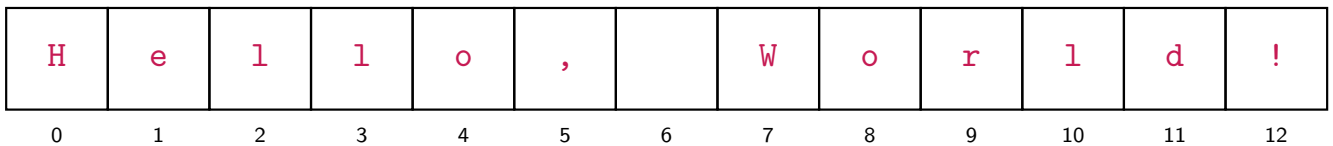



```
>>> for item in s:
  print(item, end="")

Hello, World!
```

```
>>> i = 0
>>> while(i < len(s):
  print(s[i], end="")
  i+=1

Hello, World!
```

# Strings em **Python**

- Podemos verificar se um item **pertence** a lista com o comando in:




```
>>> "e" in s
True
>>> "a" in s
False
```

- Até mesmo se uma sublista (substring) pertence a `string`:

```
>>> "llo" in s
True
```

# Strings em Python

- Mas, não podemos alterar uma posição de `string`.

```
1 >>> s = "banana"
2 >>> s[0] = "c"
3 Traceback (most recent call last):
4   File "<pyshell#7>", line 1, in <module>
5     s[0] = "c"
6 TypeError: 'str' object does not support item assignment
```

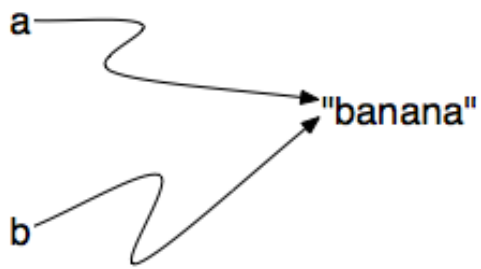
- Na verdade, `strings` em **Python** são objetos imutáveis.

# Strings em **Python**

Como `strings` em **Python** são listas imutáveis.

- **Python** otimiza recursos fazendo duas variáveis que se referem à mesma `string` se referirem ao mesmo objeto.

```
1 >>> a = "banana"
2 >>> b = "banana"
3 >>> a is b
4 True
```



# Roteiro


1. Strings em **Python**
2. Concatenação de Strings
3. Algumas operações
4. Comparação de Strings
5. Exemplos com Strings
6. Referências

# Concatenação de Strings

Para concatenar `strings`, usamos o operador `+`

```
>>> s = "Hello"
>>> x = "World";
>>> s + x
'HelloWorld'
```

- Uma nova `string` é criada, sempre.

```
>>> id(s)
139966449721520
>>> s = s + x
>>> id(s)
139966422159088
```

# Concatenação de Strings

- A concatenação não adiciona espaço como separador:

```
>>> s = "Hello"
>>> x = "World";
>>> s + " " + x
'Hello World'
```

- Podemos concatenar apenas strings

```
>>> "Hello" + 2
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    "Hello" + 2
TypeError: can only concatenate str (not "int") to str
```

  - A função str() converte um objeto em string.

# Concatenação de Strings

- O operador * faz repetições da concatenação:

```
1 >>> s = "Hello"
2 >>> s * 3
3 'HelloHelloHello'
```

- O resultado da operação é o mesmo que soma 3 vezes a `string`

```
1 >>> s + s + s
2 'HelloHelloHello'
```

# Roteiro


1. Strings em **Python**
2. Concatenação de Strings
3. Algumas operações
4. Comparação de Strings
5. Exemplos com Strings
6. Referências

# Strings maiúsculas e minúscula

Métodos `upper()`, `lower()` e `capitalize()`:

```
>>> s = "oI, bOm Dia"
>>> s.upper()
'OI, BOM DIA'
>>> s.lower()
'oi, bom dia'
>>> s.capitalize()
'Oi, bom dia'
```

- A string original `s` permanece inalterada. O método sempre retorna um novo objeto (mesmo que nenhuma mudança seja feita).

```
>>> s
'oI, bOm Dia'
>>> id(s)
139966412871344
>>> x = s.upper()
>>> id(x)
139966412870640
```

```
>>> s = 'oi, bom dia'
>>> id(s)
139966412871536
>>> x = s.lower()
>>> id(x)
139966422156144
```

# Strings maiúsculas e minúscula

- Podemos verificar se uma `string` possuí letras maiúsculas ou minúscula:

```
>>> s = "OI, BOM DIA"
>>> s.isupper()
True
>>> s.islower()
False
```

- Podemos verificar se os caracteres representam um número:

```
>>> s = "123"
>>> s.isnumeric()
True
>>> int(s)
123
```

```
>>> x = "1 23"
>>> x.isnumeric()
False
```

# Removendo caracteres

Podemos remover caracteres do *início* e *final* de uma `string`.

- O método `strip()` remove **espaços em brancos** e **pula linhas**:

```
>>> s = " \n  abc def   \n"
>>> print(s)

  abc def

>>> s.strip()
'abc def'
```

- Podemos especificar os caracteres para o método:

```
>>> x = "www.example.com"
>>> x.strip("cmowz.")
'example'
```

```
>>> x.strip(".")
'www.example.com'
>>> x.strip("w")
'.example.com'
>>> x.strip("w.")
'example.com'
```

# Substituição de substrings

O método `replace()` troca as ocorrências de uma substring por outra.

```
>>> s = "um um pão de queijo, dois dois cafés, um chocolate também."
>>> x = s.replace("um", "1")
>>> x
'1 1 pão de queijo, dois dois cafés, 1 chocolate também.'
```

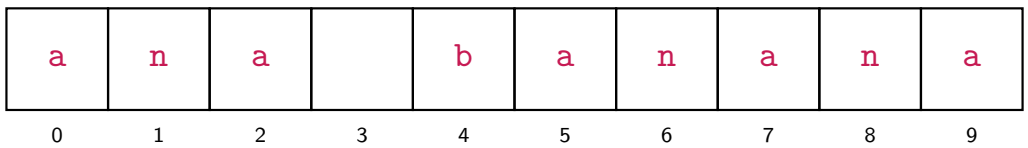
- Um terceiro parâmetro (opcional) indica quantas ocorrências queremos trocar.

```
>>> x = s.replace("um", "1", 2)
>>> x
'1 1 pão de queijo, dois dois cafés, um chocolate também.'
```

```
>>> id(s)
139966412799120
x = s.replace("abacate", "")
>>> id(x)
139966412799120 #mesma string
```

# Busca por substrings (parte 1)

O método `count()` conta o número de ocorrências não sobrepostas de uma substring.




```
>>> s = "ana banana"
>>> s.count("a")
5
>>> s.count("ana")
2
```

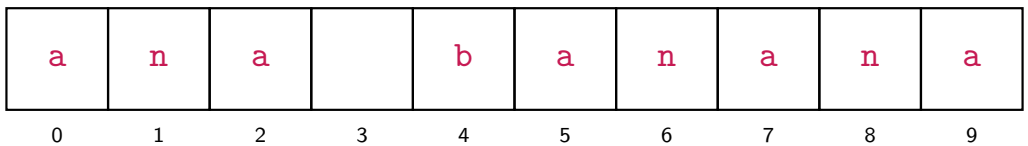
```
>>> s.count("a", 4)
3
>>> s.count("a", 4, 9)
2
```

- Podemos especificar o invervalo de busca :

  `str.count(sub, start, end)`.

# Busca por substrings (parte 2)

O método `find()` retorna a posição da 1ª ocorrência de uma substring, ou `-1` caso ela não seja encontrada.




```
>>> s = "ana banana"
>>> s.find("nana")
6
>>> s.find("x")
-1
```

```
>>> s.find("ana", 4)
5
>>> s.find("ana", 4, 7)
-1
```

- Também podemos especificar o invervalo de busca : `str.find(sub, start, end)`.

---

Para checar se uma substring ocorre em `s` é mais eficiente usar o **comando** `in`.

# Separando uma string (parte 1)

O método split(sep) separa uma string usando sep como separador.

- Retorna uma lista de substrings.

```
>>> s = "1#2#3#4#5"
>>> s.split("#")
['1', '2', '3', '4', '5']
```

- Quando sep é None ou não é fornecido:

```
>>> s = "1 2 \n  3\t4  5 "
>>> print(s)
1 2
  3        4  5
>>> s.split()
['1', '2', '3', '4', '5']
```

```
>>> l = s.split(None, 2)
>>> l
['1', '2', '3\t4  5 ']
```

- Um segundo parâmetro (opcional) indica quantos "splits" fazer.

# Separando uma string (parte 1)

Utilizamos o método split(sep) para ler listas da entrada:

- Comandos input() e split():

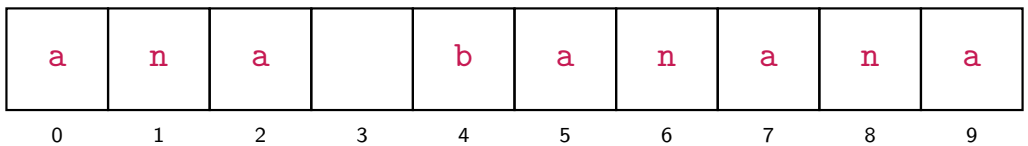
```
1 >>> s = input().split()
2 a b c 1 2 3
3 >>> s
4 ['a', 'b', 'c', '1', '2', '3']
```

- Se a entrada for uma lista de inteiros:

```
1 >>> s = [int(x) for x in input().split()]
2 1 2 3 4 5 6
3 >>> s
4 [1, 2, 3, 4, 5, 6]
```

# Separando uma `string` (parte 2)

O método `list()` transforma uma string em uma lista:




- Os itens da lista correspondem aos caracteres da string.

```
1 >>> s = "ana banana"
2 >>> list(s)
3 ['a', 'n', 'a', ' ', 'b', 'a', 'n', 'a', 'n', 'a']
```

# Unindo uma string

O método join() transforma uma lista em uma string:

```
1 >>> l = ['a', 'n', 'a', ' ', 'b', 'a', 'n', 'a', 'n', 'a']
2 >>> x = "".join(l)
3 "ana banana"
```

- O separador entre os elementos é a string utilizada no método.

```
1 >>> x
2 "ana banana"
3 >>> x = x.upper()
4 >>> x
5 'ANA BANANA'
6 >>> x = x.join(l)
7 >>> x
8 'aANA BANANAnANA BANANAaANA BANANA ANA BANANAbANA BANANAaANA BANANAnANA BAN
```

# Roteiro


1. Strings em **Python**
2. Concatenação de Strings
3. Algumas operações
4. Comparação de Strings
5. Exemplos com Strings
6. Referências

# Comparação de Strings

Os operadores de comparação também funcionam com strings:

```
>>> s = "banana"
>>> x = "banana"
>>> if(s == x):
    print("Sim, temos bananas!")
```

- **Python** é *case-sensitive*:

```
>>> s = "banana"
>>> x = "Banana"
>>> s == x
False
```

# Comparação de Strings

- Podemos comparar a ordem lexicografica de duas `strings` com os operadores >, >=, < e <=.

```
>>> s = "abacate"
>>> x = "banana"
>>> s < x
True
```

- Semelhante à ordem alfabética usada em um dicionário, exceto que letras maiúsculas vêm antes das letras minúsculas.

```
>>> s = "abacate"
>>> x = "Banana"
>>> s < x
False
```

# Comparação de Strings

- Isso porque cada caractere corresponde a um valor inteiro único.

```
>>> ord("a")
97
>>> ord("B")
66
>>> "a" < "B"
False
```

  - Quando comparamos `strings`, o **Python** converte os caracteres para seus valores ordinais e compara-os da esquerda para a direita.

# Tabela ASCII

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

# Comparação de Strings

- Há uma função inversa, `chr()` que converte inteiros para o seu caractere equivalente:

```
>>> chr(97)
'a'
>>> chr(66)
'B'
>>> print("O caractere correspondente a 100 é",chr(100),"!!!")
```

# Comparação de Strings

- Uma boa prática ao comparar `strings` é converter para um formato padrão, tal como todas as letras minúsculas:

```
1 >>> s = "banana"
2 >>> x = "Banana"
3 >>> s.lower() == x.lower()
4 True
```

# Roteiro


1. Strings em **Python**
2. Concatenação de Strings
3. Algumas operações
4. Comparação de Strings
5. Exemplos com Strings
6. Referências

# Exemplo 1: Contando o número de vogais

Escreva a função `contaVogais(frase)`, que recebe uma `string` e retorna o número de vogais presentes.

```
1 contaVogais("Estamos programando em Python")
2 >>> 9
```

# Exemplo 1: Contando o número de vogais

- Primeiro, vamos definir a função `testeVogal()`:

```
1 def testeVogal(c):
2   vogais = "aeiou"
3   if(c in vogais):
4     return True
5   else:
6     return False
```

- Precisamos deixar todas as letras minúsculas.

```
1 def contaVogais(s):
2   s = s.lower()
3   soma = 0
4   for c in s:
5     if(testeVogal(c)):
6       soma+=1
7   return soma
```

# Exemplo 2: Contando palavras

Escreva a função `contaPalavras(frase)`, que recebe uma `string` e retorna o número de palavras presentes.

```
contaPalavras("Estamos programando em Python?? Sim, com certeza.")
>>> 7
```

# Exemplo 2: Contando palavras

- Primeiro, vamos remover as pontuações:

```
1 def removePontuacao(s):
2   pontuacao = ".,:;!?"
3   for p in pontuacao:
4     s = s.replace(p, " ")
5   return s
```

- Depois, dividimos as palavras em uma `lista`:

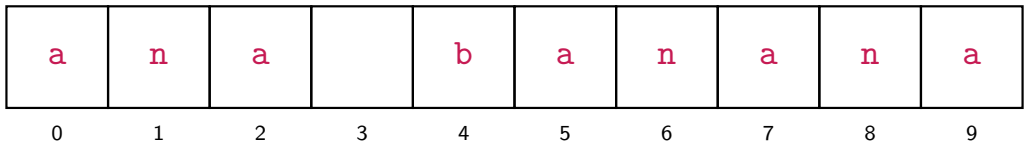
```
1 def contaPalavras(s):
2   s = removePontuacao(s)
3   l = s.split()
4   return len(l)
```

# Exemplo 3: Busca por subcadeias

Escreva a função `busca(T, p)`, que recebe duas `strings` e retorna em uma lista com todas as posições de `p` em `T`.

```
>>> busca("ana banana", "ana")
[0, 5, 7]
```

# Exemplo 3: Busca por subcadeias



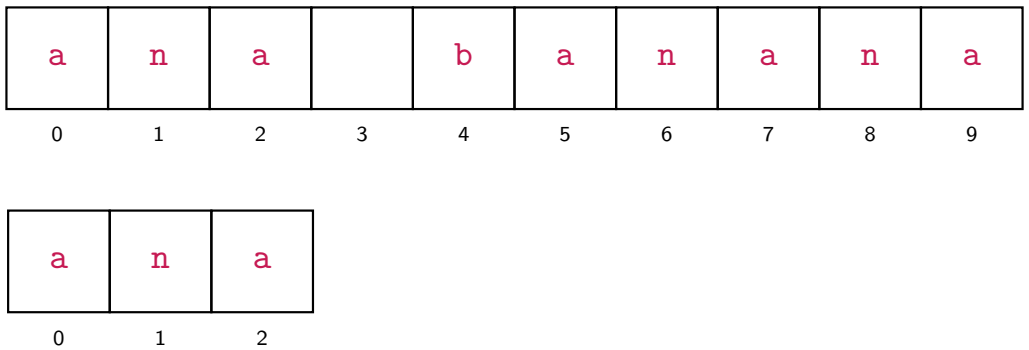

- Vimos que o método `count()` conta apenas ocorrências não sobrepostas:

```
1 >>> T = "ana banana"
2 >>> T.count("ana")
3 2
```

- E o método `find()` retorna a posição apenas da 1ª ocorrência.

```
1 >>> T = "ana banana"
2 >>> T.find("ana")
3 0
```

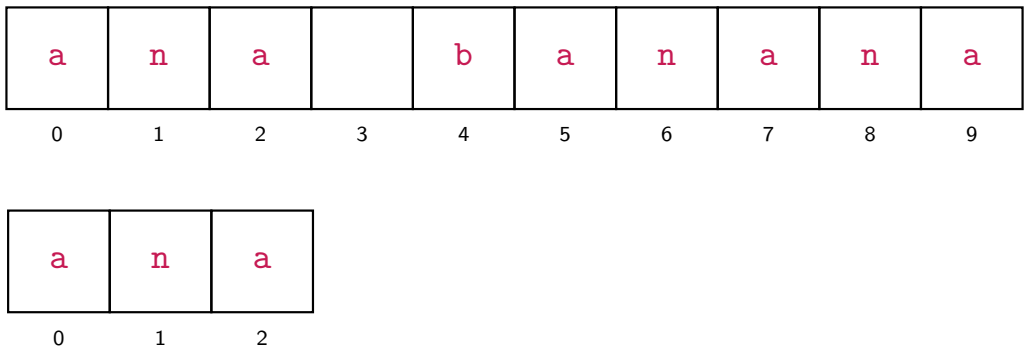
# Exemplo 3: Busca por subcadeias




- Podemos percorrer a `string T` e ir comparando com `p`:

```
1  def buscaVersao1(T, p):
2    l = []
3    n = len(T)
4    m = len(p)
5    for i in range(n-m+1):
6      count = 0
7      for j in range(m):
8        if(T[i+j] == p[j]): count+=1
9      if(count==m): l.append(i)
10   return l
```

# Exemplo 3: Busca por subcadeias




- Outra alternativa é testar `T[i:i+len(p)] == p`:

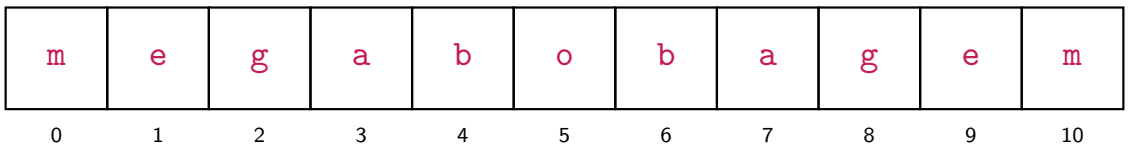
```
1  def buscaVersao2(T, p):
2    l = []
3    n = len(T)
4    m = len(p)
5    for i in range(n-m+1):
6      if(T[i:i+m] == p):
7        l.append(i)
8    return l
```

# Exemplo 4: Palindromos

Escreva a função `palindromo(texto)`, que recebe uma `string` e verifica se ela é ou não um palindromo.

```
>>> palindromo("ovo")
True
>>> palindromo("reviver")
True
>>> palindromo("mega bobagem")
True
>>> palindromo("anotaram a data da maratona")
True
```

# Exemplo 4: Palindromos

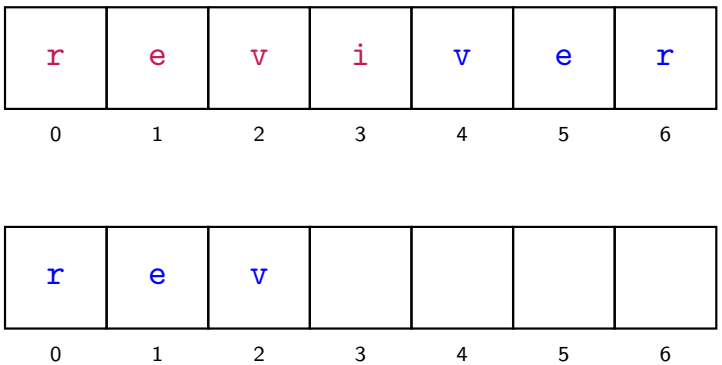



- Primeiro, removemos os espaços em branco:

```
1  def palindromoVersao1(s):
2    s = s.replace(" ", "")
3    n = len(s)
4    j = n-1
5    for i in range(n):
6      if(s[i] != s[j]):
7        break
8      j = j - 1
9    #se percorreu toda a string
10   if(j == -1):
11     return True
12   else:
13     return False
```

# Exemplo 4: Palindromos

- Outra alternativa com `slicing`:




```
1 def palindromoVersao2(s):
2     s = s.replace(" ", "")
3     r = s[::-1] #fatiamento com passo -1
4     if(s == r):
5         return True
6     else:
7         return False
```

# Exemplo 5: Data por extenso

Escreva a função `converteData(data)`, que recebe uma `string` no formato (`dd/mm/aaaa`) e retorna a data com o nome do mês por extenso.

```
>>> converteData("16/12/1982")
16 de dezembro de 1982
```

# Exemplo 5: Data por extenso

- Vamos criar uma função que converte o mês:

```
1 def retornaMes(valor):
2     meses = ["", "janeiro", "fevereiro",\
3              "março", "abril", "maio", \
4              "junho", "julho", "agosto", \
5              "setembro", "outubro", "novembro", "dezembro"]
6
7     return meses[valor]
```

- Depois, dividimos a `string` em 3 valores:

```
1 def converteData(data):
2     dia, mes, ano = data.split("/")
3     valor = int(mes)
4     return "{} de {} de {}".format(dia, retornaMes(valor), ano)
```

# Fim

Dúvidas?

# Leitura complementar

Leitura complementar:

1. https://panda.ime.usp.br/pensepy/static/pensepy/08-Strings/strings.html
2. https://wiki.python.org.br/ExerciciosComStrings

# Roteiro


1. Strings em **Python**
2. Concatenação de Strings
3. Algumas operações
4. Comparação de Strings
5. Exemplos com Strings
6. Referências

# Referências

1. Materiais adaptados dos slides do Prof. Eduardo C. Xavier, da Universidade Estadual de Campinas.