21 Reunião 21: 01/JUL/2021

Figure 1: Calvin e Hobbes de Bill Watterson

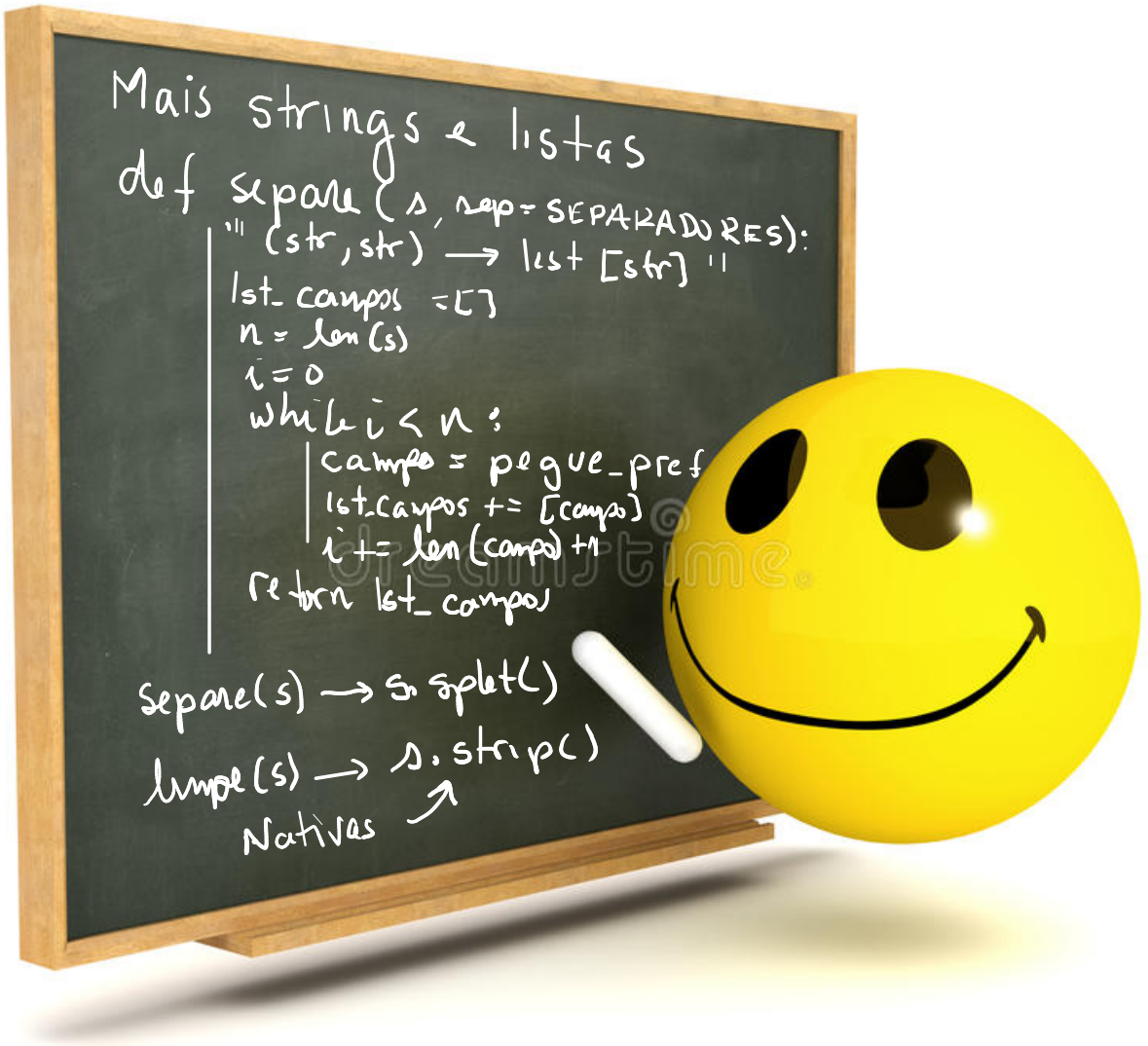
21.1 Reuniões passadas

- strings versus listas
- função `len()`
- criar strings
- acessar caracteres através de índices
- percorrer strings
- imutabilidade
- fatias: `s[ini: fim: passo]`
- fatias
- `s.strip()` (o mesmo que nosso `limpe(s)`)

Fundamental

útil para processar arquivos GIGANTESCOS

21.2 Hoje


Mais strings e listas
def separe(s, sep=SEPARADORES):
"(str,str) → lst[str]"
lst_campos = []
n = len(s)
i = 0
while i < n:
campo = pegue_pref
lst_campos += [campo]
i += len(campo)+1
return lst_campos
separe(s) → s.split()
limpe(s) → s.strip()
Nativas

21.3 Exercício : pegue maior prefixo

Escreva uma função `pegue_prefixo()` que **recebe** um índice `k`, uma string `s` e uma string `proibidos` e **retorna** a maior substring de `s` começando na posição `k` e que não possui sem caracteres na string `proibidos`. Alternativamente, podemos dizer que `pegue_prefixo()` retorna o maior prefixo da string `s[k:]` que não possui caracteres em `proibidos`.

Exemplos

```
In [5]: verdade = "MAC0110 é da hora!"

In [6]: pegue_prefixo(0, verdade, '0')
Out[6]: 'MAC'

In [7]: pegue_prefixo(0, verdade, '1')
Out[7]: 'MAC0'

In [8]: pegue_prefixo(0, verdade, ' ')
Out[8]: 'MAC0110'

In [9]: pegue_prefixo(5, verdade, 'a')
Out[9]: '10 é d'

In [10]: pegue_prefixo(5, verdade, ' a')
Out[10]: '10'

In [11]: pegue_prefixo(0, verdade, ' a')
Out[11]: 'MAC0110'

In [12]: declaracao = 'Amo estudar MAC0110!'

In [13]: pegue_prefixo(0, declaracao, '!')
Out[13]: 'Amo estudar MAC0110'
```

Solução

```
def pegue_prefixo(k, s, proibidos):
    '''(int, str, str) -> str
    RECEBE um inteiro k, uma string s e uma string proibidos.
    RETORNA a maior substring de s com inicio em k e sem caractere
        na string proibidos
    '''
    prefixo = ''
    n = len(s)
    i = k
    while i < n and s[i] not in proibidos:
        prefixo += s[i]
        i += 1
    return prefixo
```

string vazia

construção bacana

```
def pegue_prefixo(k, s, proibidos):
    '''(int, str, str) -> str
    RECEBE um inteiro k, uma string s e uma string proibidos.
    RETORNA a maior substring de s com inicio em k e sem caractere
        na string proibidos
    '''
    n = len(s)
    i = k
    while i < n and s[i] not in proibidos:
        i += 1
    return s[k:i]
```

índice deve ser válido

21.4 Exercício : leitora csv

Escreva um programa que lê um texto com **campos** (ou **valores**) separados por vírgulas e exiba cada campo ou valor em uma linha.

O programa está parcialmente feito. Você deve fazer apenas a função `separe()`.

Exemplos `separe()`

```
In [2]: s = "Como é, bom ,estudar ,MAC0110!"

In [3]: separe(s, ',')
Out[3]: ['Como é', ' bom ', 'estudar ', 'MAC0110!']

In [4]: separe(s)
Out[4]: ['Como é', ' bom ', 'estudar ', 'MAC0110!']

In [5]: separe(s, ' ')
Out[5]: ['Como', 'é,', 'bom', ',estudar', ',MAC0110!']

In [7]: separe("a,b,c,d,e\n", ',')
Out[7]: ['a', 'b', 'c', 'd', 'e\n']

In [8]: separe("a,b,c,d,e", ',')
Out[8]: ['a', 'b', 'c', 'd', 'e']

In [9]: separe("nome,idade,altura,peso,imc", ',')
Out[9]: ['nome', 'idade', 'altura', 'peso', 'imc']
```

Solução

```
BRANCO      = " \n\t\v\f"
SEPARADORES = ","

#-----------------------------------------------------------------
def main():
    '''
    Programa que lê um texto no formato csv e exibe cada campo/val
    do arquivo em uma linha.
    '''
    # 1 leia a string
    txt = input("Digite um texto csv: ")
    print(f"texto='{txt}'")

    # 2 pegue a lista de strings com os campos
    lista_str = separe(txt) #

    # 3 mostre cada campo em uma linha
    for i in range(len(lista_str)):
        print(f"{i:2}: '{lista_str[i]}'") #
```

```
def separe(txt, sep = SEPARADORES):
    ''' (str, str) -> list
    RECEBE uma string `txt` e uma string `sep`.
    RETORNA uma lista contendo os campos de txt considerando
       os caracteres em sep como separadores.
    A função "corta" o txt nos separadores.
    '''
    lst_campos = []
    n = len(txt)
    i = 0
    while i < n:
        campo = pegue_prefixo(i, txt, sep)
        lst_campos += [campo]
        i += len(campo) + 1
    return lst_campos

#-------------------------------------------------------------
def pegue_prefixo(k, s, proibidos):
    '''(int, str, str) -> str
    RECEBE um inteiro k, uma string s e uma string proibidos.
    RETORNA a maior substring de s com inicio em k e sem caractere
        na string proibidos
    '''
    prefixo = ''
    n = len(s)
    i = k
    while i < n and s[i] not in proibidos:
        prefixo += s[i]
        i += 1
    return prefixo

if __name__ == "__main__":
    main()
```

Versão *de luxe*

Escreva um programa que leia um arquivo no formato csv e exiba cada campo do arquivo em uma linha.

Em um arquivo no **formato csv** cada linha é um registro de dados. Cada registro consiste em um ou mais campos ou valores separados por uma vírgulas O uso da vírgula como separador de campos é razão do nome do formato *Comma-Separated Values*.

Um arquivo no formato csv normalmente armazena dados tabulados e sem formatação; nesse caso, cada linha terá o mesmo número de campos. Frequentemente arquivos no formato csv são usados por cientistas de dados e são bem grandes. Veja por exemplo o arquivo geographic-distribution-covid-19-cases-worldwide.csv que copiamos no ano passado do *European Centre for Disease Prevention and Control*.

```
  0   1   2 ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| n | o | m | e | , | i | d | a | d | e | , | p | e | s | o |\n |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Exemplos programa

```
Digite um texto csv: Como , é, bom , estudar, MAC0110!
texto='Como , é, bom , estudar, MAC0110!'
 0: 'Como '
 1: ' é'
 2: ' bom '
 3: ' estudar'
 4: ' MAC0110!'

Digite um texto csv: MAC0110, é, massa!
texto='MAC0110, é, massa!'
 0: 'MAC0110'
 1: ' é'
 2: ' massa!'

Digite um texto csv: MAC0110  ,  é  , da , hora!
texto='MAC0110  ,  é  , da , hora!'
 0: 'MAC0110  '
 1: '  é  '
 2: ' da '
 3: ' hora!'
```

Solução *de luxe*

```
SEPARADORES = ","

#-----------------------------------------------------------------
def main():
    '''
    Programa que lê um texto no formato csv e exibe cada campo/val
    do arquivo em uma linha.
    '''
    # Leitura de arquivo em quatro passos
    # 1 leia o nome do arquivo
    nome_arq = input("Digite o nome do arquivo csv: ")
    # 2 abra o arquivo
    arq = open(nome_arq, "r", encoding="utf-8")
    # 3 leia todo o conteudo do arquivo
    txt = arq.read()
    # 4 feche o arquivo
    arq.close()

    # exiba conteudo do arquivo
    print(f"Texto\n'{txt}'")

    # pegue a lista da strings com os campos
    lista_str = separe(txt) # o mesmo que separe(txt, SEPARADORES)

    # 3 mostre cada campo em uma linha
    for i in range(len(lista_str)):
        print(f"{i:2}: '{lista_str[i]}'") #
```

```
#-----------------------------------------------------
def separe(txt, sep = SEPARADORES):
    ''' (str, str) -> list
    RECEBE uma string `txt` e uma string `sep`.
    RETORNA uma lista contendo os campos de txt considerando
       os caracteres em sep como separadores.
    A função "corta" o txt nos separadores.
    '''
    lst_campos = []
    n = len(txt)
    i = 0
    while i < n:
        campo = pegue_prefixo(i, txt, sep)
        lst_campos += [campo]
        i += len(campo) + 1
    return lst_campos

#-----------------------------------------------------------
def pegue_prefixo(k, s, proibidos):
    '''(int, str, str) -> str
    RECEBE um inteiro k, uma string s e uma string proibidos.
    RETORNA a maior substring de s com inicio em k e sem caractere
        na string proibidos
    '''
    prefixo = ''
    n = len(s)
    i = k
    while i < n and s[i] not in proibidos:
        prefixo += s[i]
        i += 1
    return prefixo

#-----------------------------------------------
if __name__ == "__main__":
    main()
```

21.5 Mutação

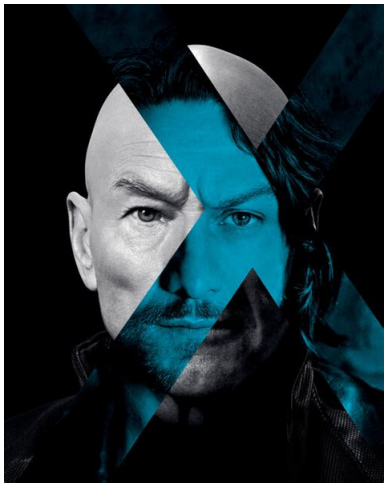


Figure 2: Xavier

Mutabilidade: listas são **mutáveis**; *podemos* alterar um componente:

```
lst[i] = x # :-)
```

Imutabilidade: strings são **imutáveis**; *não podemos* alterar um componente:

```
s[i] = 'x' # ERRO :-\
```

21.6 Strings

Strings são uma sequência de caracteres delimitadas por `"` ou `'`.

```
In [13]:  s = "Como é bom estudar MAC0110!"

In [14]: s
Out[14]: 'Como é bom estudar MAC0110!'

In [15]: print(s)
Como é bom estudar MAC0110!
```

Caractere `'\n'`

Caracteres ao sempre exibidos por `print()` exibem símbolos gráfico e **efeitos não gráficos**.

```
 0   1   2
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| l | i | n | h | a | 0 |\n | l | i | n | h | a | 1 |\n | l | i | n | h | a | 2 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Tipos nativos: `int`, `float`, `bool`, `str`, `list`, `Nonetype`

`int`, `float`, `bool` e `NoneType` são tipos de dados **primitivos**, pois seus valores *não são* compostos de partes menores. Eles não podem ser "quebrados".

Strings (`str`) e listas (`list`) são diferentes pois são compostos de partes menores.

Um **caractere** é um símbolos gráficos como letras, pontuação, espaços que são exibidos na tela. Exemplos de caracteres diferentes de letras: `' '`, `'\n'`, `'*'`, `'-'`, ...

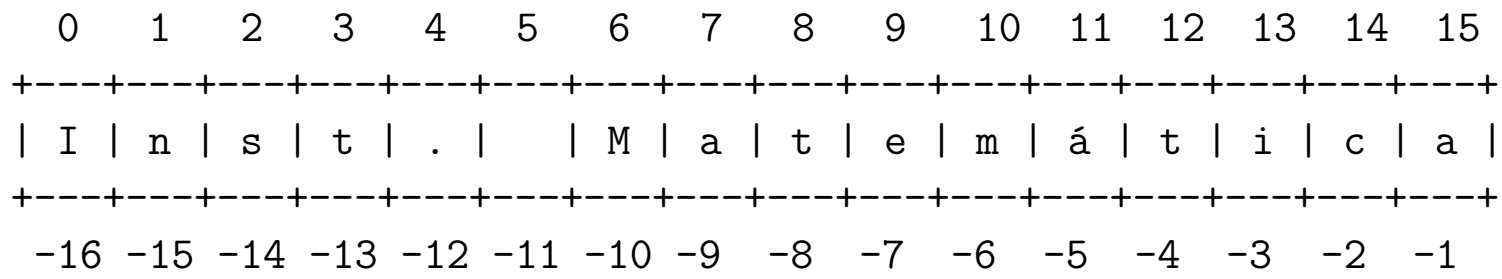
Os componentes de uma *lista* podem ser **qualquer** *coisa*.
Os componentes de *strings* são *somente* **caracteres**.

A manipulação de listas e strings é muito semelhante.
Há, no entanto, uma diferença fundamental: strings são **imutáveis**!.

Tipos formados por partes menores são chamados de **coleção de tipos de dados**.

Dependendo do que fizermos desejamos tratar uma coleção de tipos como uma única entidade ou desejamos acessar as suas partes.

Esta ambiguidade pode ser útil.



Para criar uma **string vazia** fazemos

```
In [9]:  string_vazia = ""

In [10]: string_vazia
Out[10]: ''

In [11]:  string_vazia = ''
In [12]: string_vazia
Out[12]: ''
```

Imutabilidade

Strings são **imutáveis**, não podemos alterar seus componentes

```
In [18]: s = "MAC0110"

In [19]: s[2] = 'T'    # ERRO
---------------------------------------------------------------
TypeError                                 Traceback (most recent
<ipython-input-19-32c1c57e2e7a> in <module>
----> 1 s[2] = 'T'

TypeError: 'str' object does not support item assignment
```

Não alteramos strings, **contruímos novas** strings.

```
In [23]: s = 'MAC0110'

In [24]: t = s[:2] + 'T' + s[3:]

In [25]: t
Out[25]: 'MAT0110'
```

Criar strings com *f-strings*

```
In [29]: s = f"n = {123}"

In [30]: s
Out[30]: 'n = 123'

In [31]: print(s)
n = 123

In [32]: t = f"pi = {3.14}"

In [33]: t
Out[33]: 'pi = 3.14'


In [37]: nome = 'Maria'

In [38]: w = f"nome = {nome}"

In [39]: w
Out[39]: 'nome = Maria'

In [40]: k = 123

In [41]: s = f"n = {k}"
```

```
In [42]: pi = 3.14

In [43]: t = f"pi = {pi}"

In [44]: s
Out[44]: 'n = 123'

In [45]: t
Out[45]: 'pi = 3.14'
```

Conversão para `str`

Para **converter** um objeto para uma string utilizamos a função nativa `str()`.

```
In [46]: s = str(123)

In [47]: s
Out[47]: '123'

In [48]: s = str(True)

In [49]: s
Out[49]: 'True'

In [50]: s = str(None)

In [51]: s
Out[51]: 'None'

In [52]: print(s)
None

In [53]: s = str(3.1415926)

In [54]: s
Out[54]: '3.1415926'
```

```
In [55]: print(s)
3.1415926

In [56]: s = str(True)

In [57]: s
Out[57]: 'True'

In [58]: s = str([1,2,3])

In [59]: s
Out[59]: '[1, 2, 3]'
```

Percorrer strings

Há várias maneiras e **percorrermos** uma string.

```
In [58]: s = "MAC0110!"
In [58]: i = 0
In [59]: while i < len(s):
    ...:     print(s[i])
    ...:     i + = 1
    ...:
M
A
C
0
1
1
0
!

In [60]: s = "MAC0110!"
In [61]: for i in range(len(s)):
    ...:     print(s[i])
```

```
    ...:
M
A
C
0
1
1
0
!

In [62]: s = "MAC0110!"
In [63]: for car in s:
    ...:     print(car)
    ...:
M
A
C
0
1
1
0
!
```

Algumas operações sobre strings

Strings são sequências de caracteres/símbolos delimitados por `"` ou `'`.

A seguir está uma tabela resumindo algumas das operações sobre strings. No que segue `s` e `t` são strings e `ini`,`fim`, `passo` e `n` são números inteiros.

Operação	Resultado
`t in s`	`True` se `t` é substring de `s`, senão `False`
`t not in s`	`False` se `t` é substring de `s`, senão `True`
`s + t`	a *concatenação* se `s` e `t`
`s * n` ou `n * s`	equivalente a `s` ser concatenada `n` vezes
`s[i]`	i-ésimo caractere de `s`
`s[ini:fim]`	fatia de `s` de `ini` a `fim`
`s[ini:fim:passo]`	fatia de `s` de `ini` a `fim` com passo `passo`
`len(s)`	comprimento de `s`
`s == t`	`True` se `s` e `t` são iguais, senão `False`
`s is t`	strings são **imutabilidade**, equivalente a `s == t`
`s[i]='x'`	BUMM!, ERRO, strings são **imutáveis**

21.7 Listas (`list`)

Listas são uma sequência de objetos delimitados por `[` e `]` e separados por vírgulas.

Para criar uma lista vazia fazemos

```
In [9]:  lst_vazia = []

In [10]: lst_vazia
Out[10]: []
```

Criar listas

Para cria uma nova lista fazemos

```
In [13]: lst = ["M", 'A', True, None, 3.14, 123]

In [14]: lst
Out[14]: ['M', 'A', True, None, 3.14, 123]

In [15]: print(lst)
["M", 'A', True, None, 3.14, 123]
```

Mutabilidade

Listas ***são mutáveis***, *podemos* alterar seus componentes ou acrescentar alguns

```
In [18]: lst = ["M", 'A', True, None, 3.14, 123]

In [19]: lst[2] = 'T' # altera

In [20]: lst += ["x-(wo)men"] # acrescenta

In [21]: lst
Out[21]: ['M', 'A', 'T', None, 3.14, 123, 'x-(wo)men']
```

Há várias maneiras de percorrermos uma lista.

```
In [58]: lst = [1, 2, True]
In [58]: i = 0
In [59]: while i < len(lst):
    ...:     print(lst[i])
    ...:     i + = 1
    ...:
1
2
True

In [60]: lst = [1, 2, True]
In [61]: for i in range(len(lst)):
    ...:     print(lst[i])
    ...:
1
2
True

In [62]: s = "MAC0110!"
In [63]: for item in lst:
    ...:     print(item)
    ...:
1
2
True
```

Algumas operações sobre listas

Listas são uma sequência de objetos delimitados por `[` e `]` e separados por vírgulas.

A seguir está uma tabela resumindo algumas das operações sobre listas. No que segue `lst`, `lstA` e `lstB` são listas, `x` é uma coisa qualquer e `ini`,`fim`, `passo` e `n` números inteiros.

Operação	Resultado
`x in lst`	`True` se `x` é item de `lst`, senão `False`
`x not in lst`	`False` se `x` é item de `lst`, senão `True`
`lstA + lstB`	a *concatenação* se `lstA` e `lstB`
`lst * n` ou `n * lst`	equivalente a `lst` ser concatenada `n` vezes
`lst[i]`	i-ésimo caractere de `lst`
`lst[ini:fim]`	fatia de `lst` de `ini` a `fim`
`lst[ini:fim:passo]`	fatia de `lst` de `ini` a `fim` com passo `passo`
`len(lst)`	comprimento/número de itens de `lst`
`lstA == lstB`	`True` se `lstA` e `lstB` são iguais, senão `False`
`lstA is lstB`	`True` se `lstA` e `lstB` são **apelidos** para uma **mesma** lista
`lst[i]=x`	lista são **mutáveis**