

Universidade Federal de Pernambuco
Centro de Informática

Strings e Listas

Prof. Roberto Souto Maior de Barros
roberto@cin.ufpe.br

Strings



- São seqüências de caracteres.
 - Em Python, o nome do tipo é ‘**str**’ – em geral: *string*.
- Em Python, *strings* podem ser delimitados tanto por aspas (“) quanto por apóstrofos (‘).
 - Em algumas outras linguagens somente por aspas (apóstrofo usado para um só caracter).
- Em Python, *strings* são **objetos imutáveis!**
 - Não é possível *adicionar*, *remover* ou *modificar* parte de um *string*.
 - Para isto, é sempre necessário criar um *outro string*.

Strings



- Relembrando...
 - Usar `\`, `'` e `\\` para inserir o símbolo após a `\`.
 - Usar `\n` para mudar de linha (na impressão).
 - Usar `\t` para dar um *<tab>* (na impressão).
- Exemplos:
 - fruta = 'banana'
 - mensagem = "Digite um número: "
 - sigilo = "Classificado como 'top secret'!"
 - Texto = 'Digite o seu "sobrenome" antes do "nome".'
 - Linha = 'Imprimir símbolos `\`, `\t \`, e `\t \\ \n`'

Strings - comparação



- Operadores de comparação funcionam também para *strings*!
 - Não só `==` e `!=`, também `<`, `>`, `<=` e `>=`.
 - A ordem do sistema interno de representação é usada (ASCII, Unicode, etc.).
 - Ordem alfabética dentro do mesmo tipo de letra.
 - Mas, as maiúsculas são **menores** que as minúsculas.
- **Exemplos:**
 - `'B' < 'a'` é True.
 - `'b' < 'banana'` também é True.

Strings – acesso usando índices



- Cada um dos caracteres de um *string* pode ser acessado por um índice posicional:
 - Em Python, estes índices (posições) começam a ser contados em **zero** e são escritos entre colchetes.
 - Só em Python, índices negativos a partir de **-1** podem ser usados para acessá-los de traz para a frente.
 - Usar um índice fora do intervalo válido de acordo com o tamanho do *string* causa um erro fatal! – ***IndexError***.
- **OBS:** Porque *strings* em Python são *imutáveis*, *não é* permitido usar comando de atribuição para uma posição de um string usando um índice.



Strings – acesso usando índices



- Exemplos:

- fruta = 'bananas!'
- primLet = fruta[0] # Resultado é 'b'.
- ultLet = fruta[-1] # Resultado é '!'.
- penuLet = fruta[-2] # Resultado é 's'.
- fruta [7] = '.' # Isto não é permitido – erro fatal!
- fruta = fruta[:7] + '.' # Ok. Outro objeto é criado.
Resultado é 'bananas.'



Strings – comando `for`



- Em Python, o comando `for` pode ser usado para acessar os caracteres de um *string* “um a um” *sem precisar usar índices!*

Sintaxe do comando `for` para percorrer *strings*.

```
for ch in qualquerString :  
    comandoUsandoCh
```

- **Exemplo:**

```
fruta = 'banana'
```

```
for c in fruta : # c recebe cada caracter da variável fruta.  
    print (fruta, c) # c assume cada uma das letras...
```

Strings – operadores e funções



- A maioria das linguagens de programação tem operadores e/ou funções que visam facilitar a manipulação de *strings*.
- Em Python temos:
 - **len**: tamanho (quantidade de caracteres) do *string*.
 - **+**: concatenação de *strings* (para juntar/emendar...).
 - Conversão de valores de outros tipos não é automática.
 - *****: repetição de *strings* – concatenação repetida...
 - **in**: indica se um *string* está contido em outro.
 - ****: permite que um *string* muito longo possa continuar a ser digitado na linha seguinte do programa.



Strings – operadores e funções



- Exemplos:
 - fruta = 'banana'
 - tam = len (fruta) # Resultado é 6.
 - qtd = 10
 - fruta2 = fruta + ' comprida ' + str (qtd)
 - spam = 'Spam!' * 3 # spam = 'Spam!Spam!Spam!'
 - res = fruta in fruta2 # res ficará com True.
 - temA = 'a' in fruta # temA também ficará com True.
 - alfabetos = "ABCDEFGHIJKLMNOPQRSTUVWXYZ\abcdefghijklmnopqrstuvwxyz"



Strings – interpolação



- O operador de interpolação (%) (só em Python):
 - é uma versão mais *eficiente* de concatenação.
 - é geralmente usado para combinar/intercalar valores de variáveis dentro de um *string*.
 - é útil também em comandos *print* para formatar saída.
 - usa “máscaras” para marcar os locais onde inserir os valores das variáveis – estes serão *parâmetros* para o operador %.
 - Pode-se informar um *tamanho* (em colunas - opcional) para incluir o valor do conteúdo mas, se o tamanho informado for *insuficiente*, ele será *ignorado*.

Strings – interpolação



- As máscaras são:
 - `%d`: números inteiros – *int* e *long*.
 - `%f`: números reais – *float*.
 - `%s`: *strings*.
 - O *tamanho* pode ser informado *após* o `%` e, no caso dos reais, também o número de casas decimais.
 - Ex. `%5d` ou `%8.2f` ou `%.2f`.
- Exemplos:
 - `print ('A=%d e %s.' % (a, mensagem))`
 - `print ("Média dos %3d alunos é %8.2f." % (qtd, media))`

Strings – formatação



- O operador de formatação (f) (só em Python 3):
 - é semelhante à interpolação.
 - usa as mesmas máscaras.
 - só a sintaxe é ligeiramente diferente.
- Exemplos:
 - `print (f'A = {a:2d} e {mensagem:10s}.')`
 - `print (f"Média dos {qtd:3d} alunos é {media:8.2f}.")`



Strings – substring



- Um *substring* é um operador que serve para ter acesso (ou recuperar) um **pedaço** de um *string*:
 - A maioria das linguagens tem esta funcionalidade.
 - Em Python são chamados de *slices*.
 - Na sintaxe de Python, usa-se dois índices separados por **:** (dois pontos) para significar um intervalo de índices (fechado à esquerda e aberto à direita).
 - Cada um dos limites pode ser omitido e o significado será “**a partir do início**” e “**até o fim**”, respectivamente.
 - Se o intervalo for vazio, ou seja, o segundo parâmetro for menor ou igual ao primeiro, o resultado é o string nulo – “”.



Strings – substring



- **Exemplos:**
 - fruta = 'banana'
 - silaba2 = fruta [2:4] # Resultado é 'na'.
 - silaba1 = fruta [0:2] # Resultado é 'ba'.
 - silaba1 = fruta [:2] # Mesma coisa...
 - sufixo = fruta [3:6] # Resultado é 'ana'.
 - sufixo = fruta [3:] # Mesma coisa...
 - tudo = fruta [:] # Válido mas não faz muito sentido...
 - tudo = fruta # Válido e mais eficiente.
 - vazio = fruta [3:3] # Resultado é o string nulo - "".

Strings – métodos



- Python também fornece métodos, que são como operadores mas usam uma *sintaxe diferente*.
 - Isto é consequência de *strings* serem objetos!
 - Sintaxe é: *stringQualquer.nomeMétodo* (parâmetros)
- Alguns métodos comuns para *string*:
 - *upper*: troca todas as letras para maiúsculas.
 - *lower*: troca todas as letras para minúsculas.
 - *find*: procura posição de um *substring* em um *string*.
 - *replace*: troca ocorrências de um *substring* por outro.
 - *strip*: retira espaços no início, no final ou múltiplos, além de outros *whitespaces* como *tab*, *newline*, etc.

Strings – métodos



- **Exemplos:**

- `fr = ' Banana! '`
- `fr = fr.strip()` **# Resultado é 'Banana!'**
- `fr2 = fr.upper()` **# Resultado é 'BANANA!'**
- `fr3 = fr.lower()` **# Resultado é 'banana!'**
- `pos1 = fr.find('na')` **# Resultado é 2.**
- `pos2 = fr.find('na',3)` **# Resultado é 4.**
- `pos3 = fr.find('Ba',2)` **# Resultado é -1.**
- `pos4 = fr.find('!')` **# Resultado é 6.**
- `pos5 = fr.find('!',1,5)` **# Resultado é -1.**
- `fr4 = fr.replace('a','A')` **# Resultado é 'BAnAnA!'**.
- `fr5 = fr.replace('a','A',2)` **# Resultado é 'BAnAna!'**.



Listas e Arrays



- O conceito de lista (ou seqüência) se refere a um *conjunto de valores*, normalmente de um *mesmo tipo*, sobre o qual existe a noção de *ordem*.
 - Existe o primeiro elemento (início da lista), o próximo (segundo) e assim por diante, até o último (fim da lista).
 - Conceitualmente as listas podem sofrer modificações, inclusive a inserção e remoção de elementos.
 - Ou seja, o tamanho da lista também pode mudar.
- Em geral, as linguagens *não* fornecem o tipo lista, mas um tipo mais restrito chamado de *array*.
 - Os *arrays* têm *tamanho fixo depois de criados...*



Listas e *Arrays*



- *Arrays* são variáveis compostas *homogêneas*:
 - Variáveis de um **mesmo tipo** e com um **nome único**.
 - Para se referir a cada um dos componentes usa-se um ou mais **índices** – valores inteiros.
 - Na maioria das linguagens, *arrays* têm *tamanho fixo*:
 - Em muitas, o tamanho é dado por uma constante que é escolhida já no momento da codificação do programa.
 - Ou seja, é preciso “prever” um tamanho máximo!
 - Em algumas (como Java), a escolha pode ser feita em tempo de *execução* mas, mesmo assim, o tamanho *não* pode mudar *depois da criação...*
 - Ainda pode ser preciso “prever” o tamanho máximo!



Listas e *Arrays*



- Vantagens
 - Evitar criar nomes diferentes para várias variáveis que contenham informações semelhantes.
 - Facilitar a repetição de comandos usando variáveis diferentes (os elementos de um *array*) escritos dentro de comandos de repetição, o que evita escrever várias vezes estes comandos.
- Tipos de *arrays*:
 - Unidimensionais – vetores.
 - Bidimensionais – matrizes.
 - Multidimensionais.



Listas e Arrays



- Em *Python*, ao contrário das outras linguagens:
 - Listas são implementadas nativamente – tipo *list*.
 - Inclusive com *tamanho variável*.
 - Não é fornecido o tipo *array*.
- No tipo *list* de *Python*, os elementos podem ser *heterogêneos*, ou seja, o *tipo* dos elementos *não* precisa ser único. Porém, isto *não tem* muita utilidade na prática (no *mundo real*)...
- **OBS:** Minha recomendação é usá-lo, sempre que possível, com elementos de um *mesmo tipo*.



Listas - sintaxe



- Listas são delimitadas por *colchetes* ([e]) e os elementos são separados por *vírgulas*.
- O tipo *list* é um tipo como outro qualquer e pode ser usado sem definição prévia.
- Listas também são implementadas como *objetos*.
- Exemplos:
 - lista1 = [10,20,30,40]
 - lista2 = ['a', 'b', 'c']
 - lista3 = [] # Lista vazia (sem elementos).
 - lista4 = [None, None] # Lista com elementos nulos.



Listas – acesso usando índices



- Como nos *strings*, cada um dos elementos de uma lista pode ser acessado por um índice posicional, com as mesmas regras:
 - Os índices (posições) começam a ser contados em **zero** e são escritos entre colchetes.
 - Só em Python, índices negativos a partir de **-1** podem ser usados para acessá-los de traz para a frente.
 - Usar um índice fora do intervalo válido de acordo com o tamanho da lista causa um erro fatal! – ***IndexError***.
 - Esta é uma limitação (quase) universal também no caso das outras linguagens de programação e dos *arrays*.

Listas – acesso usando índices



- Porque, *diferentemente dos strings*, as listas *podem* ser modificadas, *não há restrições* à alteração do valor de um elemento da lista usando o comando de atribuição e um índice.
- **Exemplos:**
 - lista = [10, 20, 30, 40]
 - prim = lista [0] # Resultado é 10.
 - ult = lista[-1] # Resultado é 40.
 - penult = lista [-2] # Resultado é 30.
 - lista [1] = 50 # Resultado é [10, 50, 30, 40].



Listas – operadores e funções



- Operadores e/ou funções para manipular listas:
 - **len**: tamanho (quantidade de elementos) da lista.
 - **+**: concatenação de listas (para juntar/emendar...).
 - *****: repetição de elementos – concatenação repetida...
 - Também é aceito para repetição de listas mas não é recomendado porque resultado *não é* o que parece...
 - **in**: indica se um *elemento* está contido em uma lista.
 - **del**: deleta elemento pelo índice, diminuindo a lista.
 - **min**: menor elemento da lista.
 - **max**: maior elemento da lista.
 - **sum**: soma dos elementos da lista.



Listas – operadores e funções



- Exemplos:

- lista = [10, 20, 30, 40]
- tam = len (lista) # Resultado é 4.
- lis2 = lista + [50] # Resultado é [10, 20, 30, 40, 50].
- lis3 = [0] * 50 # Resultado é [0, 0, 0, ..., 0].
- res = 30 in lista # res ficará com *True*.
- res = [30] in lista # res ficará com *False*.
- del lis2 [3] # Resultado é [10, 20, 30, 50].
- men = min (lista) # Resultado é 10.
- mai = max (lista) # Resultado é 40.
- total = sum (lista) # Resultado é 100.



Listas – comando `for`



- Em Python, o comando `for` pode ser usado para acessar os elementos de uma lista “um a um” *sem precisar usar índices!* (como nos *strings...*)

Sintaxe do comando `for` para listas (sem usar índices).

`for elem in qualquerLista :`

comandoUsandoElem

- **Exemplo:**

```
lista = [10, 20, 30, 40]
```

```
for e in lista : # e recebe cada elemento de lista.
```

```
    print (e) # e assume 10, 20, 30 e finalmente 40.
```



Listas – comando `for`



- Mas também é possível acessar e/ou alterar os elementos de uma lista usando o comando `for` mais tradicional, percorrendo pelos *índices*!
 - Esta é a única maneira possível na maioria das LP...

- **Exemplo:**

```
lista = [10, 20, 30, 40]
```

```
qtd = len(lista) # Se o valor não estiver disponível.
```

```
for i in range(qtd): # i recebe somente os índices...
```

```
    lista[i] = lista[i] * 2 # e o acesso é assim...
```

```
print(lista) # lista ficará com [20, 40, 60, 80].
```

Listas – slices



- Em Python, os *slices* também são válidos para ter acesso ou recuperar um **pedaço** de uma lista, com as mesmas regras usadas com *strings*:
 - Usa-se dois índices separados por **:** (dois pontos) para significar um intervalo de índices (fechado à esquerda e aberto à direita).
 - Cada um dos limites pode ser omitido e o significado será “**a partir do início**” e “**até o fim**”, respectivamente.
 - Se o intervalo for vazio, ou seja, o segundo parâmetro for menor ou igual ao primeiro, o resultado é uma lista vazia – **[]**.

Listas – slices



- Exemplos:
 - lista = [10, 20, 30, 40, 50, 60]
 - ex1 = lista [1:3] # Resultado é [20, 30].
 - ex2 = lista [0:2] # Resultado é [10, 20].
 - ex2 = lista [:2] # Mesma coisa...
 - ex3 = lista [3:6] # Resultado é [40, 50, 60].
 - ex3 = lista [3:] # Mesma coisa...
 - nada = lista [1:1] # Resultado é a lista vazia – [].
 - copia = lista [:] # Cria outra lista igual.
 - mesma = lista # Novo nome para *mesmo objeto*...



Sobre *Tipos* (fazendo um aparte)



- Nas linguagens de programação (LP), em geral, há dois conjuntos de tipos:
 - Tipos **básicos** (ou primitivos):
 - Alocação **estática** de memória: feita já na *compilação*.
 - As variáveis se referem diretamente às posições de memória usadas para armazenar os dados.
 - Tipos **referência** (inclui os “objetos” em LP OO...):
 - Alocação **dinâmica** de memória: pode também ser feita (e em geral é) em tempo de *execução*.
 - As variáveis só armazenam os endereços de memória (“ponteiros”) onde os dados estão armazenados.
- **OBS:** Em Python, *todos* os tipos são **dinâmicos**...



Listas – métodos



- Alguns métodos básicos para manipular listas:
 - **append**: insere um *elemento* no *final* da lista.
 - **extend**: semelhante, mas concatena uma outra *lista*.
 - **insert**: insere um *elemento* num dado *índice* – desloca os elementos para a direita para abrir espaço.
 - **remove**: remove primeira ocorrência de *elemento*. (*)
 - **index**: retorna *índice* da prim. ocorr. de *elemento*. (*)
 - **reverse**: inverte a ordem dos elementos da lista.
 - **count**: retorna *número* de ocorr. de *elemento* na lista.
- (*) Se o elemento não existir, ocorre um erro fatal!



Listas – métodos



- Exemplos:
 - lista = [20] # Cria uma lista com um só elemento.
 - lista.append(40) # Resultado é [20, 40].
 - lista.extend([50, 60]) # Resultado é [20, 40, 50, 60].
 - lista.remove(50) # Resultado é [20, 40, 60].
 - lista.remove(70) # Causa erro fatal!
 - lista.insert(1, 30) # Resultado é [20, 30, 40, 60].
 - pos = lista.index(40) # Resultado é 2.
 - pos = lista.index(80) # Causa erro fatal!
 - lista.reverse() # Resultado é [60, 40, 30, 20].
 - num = lista.count(20) # Resultado é 1.



Listas – métodos



- Outros métodos para manipular listas:
 - **sort**:
 - Ordena uma lista de várias formas...
 - Em ordem crescente ou decrescente dos elementos.
 - Pode também usar uma função de comparação definida pelo usuário...
 - Pode ainda ordenar pelos resultados de uma função qualquer aplicada aos elementos da lista.
 - **pop**:
 - Remove um *elemento* da lista baseado no valor de um *índice* e o retorna como resultado.
 - Se omitir o índice, remove o *último* elemento da lista.

Listas – métodos



- Exemplos:
 - lista = [30, 20, 50, 10, 40]
 - lista.sort () # Resultado é [10, 20, 30, 40, 50].
 - lista.sort(reverse=True) # Res. é [50, 40, 30, 20, 10].
 - lista.sort (None, None, True) # Mesma coisa...
 - web = ['www', 'ufpe', 'br']
 - web.sort (key=len) # Res. é ['br', 'www', 'ufpe']
 - web.sort (None, len) # Mesma coisa...
 - r1 = lista.pop () # r1 = 10 e lista = [50, 40, 30, 20].
 - r2 = lista.pop (1) # r2 = 40 e lista = [50, 30, 20].
 - r3 = lista.pop (5) # Causa erro fatal!



Algumas observações...



- Os elementos de uma lista podem ser atribuídos a variáveis separadas, usando a atribuição múltipla.
 - O número de variáveis utilizado deve ser exatamente igual ao tamanho da lista.
 - Por exemplo, se a variável `lista` estiver com `[10, 20, 30]`, é possível fazer a atribuição: `a, b, c = lista`.
- Em muitos problemas, o tamanho necessário para uma lista é *fixo e conhecido* no início.
 - Nestes casos, é interessante/recomendado criar a lista já com o tamanho correto, porque é bem mais eficiente. Por exemplo: `lista1 = [None]*qtd` ou `lista2 = [0]*100`.



Listas e *strings*



- Funções e métodos envolvendo listas e *strings*:
 - **list**: função que transforma um *string* em uma *lista* de caracteres.
 - **split**: método que divide um *string* em uma lista de *strings* menores. Um segundo *string*, recebido como parâmetro, serve para definir os pontos de separação e não será parte do conteúdo da lista resultante.
 - **join**: método que insere um *string* como separador no resultado da transformação de uma *lista de strings* em um único *string*.
 - Se for aplicado ao *string* nulo, o resultado é a simples concatenação dos elementos da lista.



Listas – métodos



- Exemplos:
 - fruta = 'Banana'
 - fr1 = `list` (fruta) # Resultado é ['B', 'a', 'n', 'a', 'n', 'a'].
 - web = 'www.ufpe.br'
 - web1 = web.`split` ('.') # Resultado é ['www', 'ufpe', 'br'].
 - web2 = web.`split` (':') # Resultado é ['www.ufpe.br'].
 - fr2 = "-".`join` (fr1) # Resultado é "B-a-n-a-n-a".
 - fr3 = "".`join` (fr1) # Resultado é "Banana".
 - web2 = '\$'.`join` (web1) # Resultado é 'www\$ufpe\$br'.
 - num = [10, 20, 30, 40]
 - erro = ".`join` (num) # Causa erro fatal!

Arrays multidimensionais?



- Já vimos que Python não tem o tipo *array*.
 - Mas os elementos de uma lista podem ser outras listas. Isto permite “simular” matrizes e outros tipos de *array*. Porém, para mais de 2 dimensões é *ruim/complicado*!
- **OBS importante:** *Não use* a notação de repetição para criar *arrays* multidimensionais pois ela **não funcionará** da maneira correta...
 - as dimensões mais externas ficarão apenas com cópias dos endereços, ao invés de objetos independentes...
 - Será preciso usar um ou mais comandos **for** para criar adequadamente a estrutura desejada.

Matrizes



- Exemplo:

```
m = [ ] # Cria uma lista vazia
```

```
for i in range (3) : # Cria matriz 3x4.
```

```
    m.append ([0]*4)
```

```
# Agora para manipular os elementos...
```

```
for i in range (3) : # O for externo será mais lento...
```

```
    for j in range (4) : # Para cada i passa por todos os j.
```

```
        m[i][j] = 10*(i+1) + j + 1
```

```
for i in range (3) : # Para imprimir como matriz...
```

```
    print (m [i])
```

