MC102 — Listas e Strings

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

Atualizado em: 2023-04-27 13:02

Listas: relembrando

Acesso por *índice* (começando em zero)

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`
- `lista = [1.3, 7.5, -2.1]`

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`
- `lista = [1.3, 7.5, -2.1]`
- `lista = [x, y, z]`

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`
- `lista = [1.3, 7.5, -2.1]`
- `lista = [x, y, z]`
- `lista = [1, "mc102", 3.7]`

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`
- `lista = [1.3, 7.5, -2.1]`
- `lista = [x, y, z]`
- `lista = [1, "mc102", 3.7]`

Outras informações:

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`
- `lista = [1.3, 7.5, -2.1]`
- `lista = [x, y, z]`
- `lista = [1, "mc102", 3.7]`

Outras informações:

- `lista.append(x)`: adiciona `x` no final da lista

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`
- `lista = [1.3, 7.5, -2.1]`
- `lista = [x, y, z]`
- `lista = [1, "mc102", 3.7]`

Outras informações:

- `lista.append(x)`: adiciona `x` no final da lista
- `lista.clear()`: remove todos os elementos

Listas: relembrando

Acesso por *índice* (começando em zero)

- `lista[0]`, `lista[1]`, ...(para leitura/escrita)

Criando uma lista:

- `lista = []` ou `lista = list()` - lista vazia
- `lista = [1, 7, 2, 2, 15]`
- `lista = ["ana", "joão", "pedro"]`
- `lista = [1.3, 7.5, -2.1]`
- `lista = [x, y, z]`
- `lista = [1, "mc102", 3.7]`

Outras informações:

- `lista.append(x)`: adiciona `x` no final da lista
- `lista.clear()`: remove todos os elementos
- `len(lista)`: número de elementos da lista

Exercícios

1. Faça uma função que devolve uma cópia de uma lista dada
2. Faça uma função que, dadas duas listas `l1` e `l2`, adiciona os elementos de `l2`, em ordem, no final de `l1`
3. Faça uma função que encontra o maior valor de uma lista de números e devolve o índice desse elemento
4. Faça uma função que, dada uma lista `l` e um valor `x` (de qualquer tipo), devolve o primeiro índice de `l` que é igual a `x` (e lança um `ValueError` se não estiver presente)
5. Faça uma função que, dada uma lista `l`, devolve uma nova lista que é `l` invertida.
   - Modifique o exercício anterior para, ao invés de devolver uma nova lista, de fato alterar `l` (*in-place*)

Outros métodos de `list`

`list` nos dá vários métodos que realizam estas tarefas:

Outros métodos de `list`

`list` nos dá vários métodos que realizam estas tarefas:

- `l.copy()`: devolve uma cópia de `l`

Outros métodos de `list`

`list` nos dá vários métodos que realizam estas tarefas:

- `l.copy()`: devolve uma cópia de `l`
- `l1.extend(l2)`: adiciona os elementos de `l2` em `l1`

Outros métodos de `list`

`list` nos dá vários métodos que realizam estas tarefas:

- `l.copy()`: devolve uma cópia de `l`
- `l1.extend(l2)`: adiciona os elementos de `l2` em `l1`
- `l.index(x)`: devolve o primeiro índice `i` de `l` tal `l[i] == x` (e lança um `ValueError` se não existir)

Outros métodos de `list`

`list` nos dá vários métodos que realizam estas tarefas:

- `l.copy()`: devolve uma cópia de `l`
- `l1.extend(l2)`: adiciona os elementos de `l2` em `l1`
- `l.index(x)`: devolve o primeiro índice `i` de `l` tal `l[i] == x` (e lança um `ValueError` se não existir)
- `l.reverse()`: inverte `l` in-place

Índices e Slices

Os índices podem ser também negativos:

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)
  - `l[i:]` - todos os elementos a partir do `i` (omitindo `f`)

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)
  - `l[i:]` - todos os elementos a partir do `i` (omitindo `f`)
  - `l[:]` - toda a lista (omitindo `i` e `f`)

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)
  - `l[i:]` - todos os elementos a partir do `i` (omitindo `f`)
  - `l[:]` - toda a lista (omitindo `i` e `f`)
  - E podemos usar valores negativos para os índices!

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)
  - `l[i:]` - todos os elementos a partir do `i` (omitindo `f`)
  - `l[:]` - toda a lista (omitindo `i` e `f`)
  - E podemos usar valores negativos para os índices!
- `l[i:f:p]` nos dá os elementos `l[r]`, onde `r = i + k * p` com `k` inteiro positivo e `r < f` (como no `range`)

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)
  - `l[i:]` - todos os elementos a partir do `i` (omitindo `f`)
  - `l[:]` - toda a lista (omitindo `i` e `f`)
  - E podemos usar valores negativos para os índices!
- `l[i:f:p]` nos dá os elementos `l[r]`, onde `r = i + k * p` com `k` inteiro positivo e `r < f` (como no `range`)
- Você pode usar slice para:

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)
  - `l[i:]` - todos os elementos a partir do `i` (omitindo `f`)
  - `l[:]` - toda a lista (omitindo `i` e `f`)
  - E podemos usar valores negativos para os índices!
- `l[i:f:p]` nos dá os elementos `l[r]`, onde `r = i + k * p` com `k` inteiro positivo e `r < f` (como no `range`)
- Você pode usar slice para:
  - copiar parte da lista (ex: `l2 = l1[::2]`)

Índices e Slices

Os índices podem ser também negativos:

- `lista[-1]` é o último elemento
- `lista[-2]` é o penúltimo elemento
- e, assim por diante, até
- `lista[-len(lista)]` que é o primeiro elemento

E podemos fatiar (*slice*) a lista:

- `l[i:f]` nos dá os elementos `l[i]`, `l[i + 1]`, ..., `l[f - 1]`
  - `l[:f]` - até o elemento `f - 1` (omitindo `i`)
  - `l[i:]` - todos os elementos a partir do `i` (omitindo `f`)
  - `l[:]` - toda a lista (omitindo `i` e `f`)
  - E podemos usar valores negativos para os índices!
- `l[i:f:p]` nos dá os elementos `l[r]`, onde `r = i + k * p` com `k` inteiro positivo e `r < f` (como no `range`)
- Você pode usar slice para:
  - copiar parte da lista (ex: `l2 = l1[::2]`)
  - mudar valores da lista (ex: `l1[1:10] = ['x']`)

Exercícios

1. Faça uma função que, dados uma lista `l`, um índice `i`, e um objeto `x`, insere `x` de forma que ele fique no índice `i` de `l`.
   - Faça um versão com `slice`
   - E uma versão sem `slice`

Exercícios

1. Faça uma função que, dados uma lista `l`, um índice `i`, e um objeto `x`, insere `x` de forma que ele fique no índice `i` de `l`.
   - Faça um versão com `slice`
   - E uma versão sem `slice`
2. Faça uma função que, dados uma lista `l` e um índice `i`, remove e devolve o elemento de `l` no índice `i`.
   - Dica: use `slice`

Exercícios

1. Faça uma função que, dados uma lista `l`, um índice `i`, e um objeto `x`, insere `x` de forma que ele fique no índice `i` de `l`.
   - Faça um versão com `slice`
   - E uma versão sem `slice`
2. Faça uma função que, dados uma lista `l` e um índice `i`, remove e devolve o elemento de `l` no índice `i`.
   - Dica: use `slice`
3. Faça uma função que, dados uma lista `l` e um objeto `x`, remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se ele não estiver em `l`).
   - Dica: combine as funções que você já criou e o que você já sabe de `list`!

Mais métodos de `list`

Outros métodos:

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Também é possível:

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Também é possível:

- Somar duas listas: `l1 + l2` ou `l1 += l2`

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Também é possível:

- Somar duas listas: `l1 + l2` ou `l1 += l2`
  - Concatena `l1` com `l2`

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Também é possível:

- Somar duas listas: `l1 + l2` ou `l1 += l2`
  - Concatena `l1` com `l2`
- Multiplicar uma lista por um inteiro: `l1 * 2` ou `l1 *= 2`

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Também é possível:

- Somar duas listas: `l1 + l2` ou `l1 += l2`
  - Concatena `l1` com `l2`
- Multiplicar uma lista por um inteiro: `l1 * 2` ou `l1 *= 2`
  - Concatena `l1` com `l1` várias vezes

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Também é possível:

- Somar duas listas: `l1 + l2` ou `l1 += l2`
  - Concatena `l1` com `l2`
- Multiplicar uma lista por um inteiro: `l1 * 2` ou `l1 *= 2`
  - Concatena `l1` com `l1` várias vezes
- Na primeira opção, uma nova lista é criada

Mais métodos de `list`

Outros métodos:

- `l.insert(i, x)`: insere `x` antes do índice `i`
  - `x` fica no índice `i`
- `l.pop(i)`: remove o elemento de índice `i` de `l` e o devolve
  - Ou então `del l[i]`
- `l.pop()`: remove o último elemento de `l` e o devolve
- `l.remove(x)`: remove a primeira ocorrência de `x` de `l` (e lança um `ValueError` se não existir)

Também é possível:

- Somar duas listas: `l1 + l2` ou `l1 += l2`
  - Concatena `l1` com `l2`
- Multiplicar uma lista por um inteiro: `l1 * 2` ou `l1 *= 2`
  - Concatena `l1` com `l1` várias vezes
- Na primeira opção, uma nova lista é criada
- Na segunda, a lista é alterada (como no `extend`)

Strings

A classe `str` é parecida com a classe `list`

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`
  - Inclusive você pode escrever `for letra in string:`

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`
  - Inclusive você pode escrever `for letra in string:`
  - E você pode usar *slices* também

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`
  - Inclusive você pode escrever `for letra in string:`
  - E você pode usar *slices* também
- Mas você não pode alterar um caracter...

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`
  - Inclusive você pode escrever `for letra in string:`
  - E você pode usar *slices* também
- Mas você não pode alterar um caracter...
  - `s[1] = 'A'`

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`
  - Inclusive você pode escrever `for letra in string:`
  - E você pode usar *slices* também
- Mas você não pode alterar um caracter...
  - `s[1] = 'A'`
  - `TypeError: `str' object does not support item assignment`

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`
  - Inclusive você pode escrever `for letra in string:`
  - E você pode usar *slices* também
- Mas você não pode alterar um caracter...
  - `s[1] = 'A'`
  - `TypeError: `str' object does not support item assignment`
- Nem remover um caracter (`del s[1]`)

Strings

A classe `str` é parecida com a classe `list`

- Podemos acessar os caracteres da string usando índice
  - Um caracter é simplesmente uma string de tamanho 1
  - Ex: `s[1]` é o segundo caracter de `s`
  - Inclusive você pode escrever `for letra in string:`
  - E você pode usar *slices* também
- Mas você não pode alterar um caracter...
  - `s[1] = 'A'`
  - `TypeError: `str' object does not support item assignment`
- Nem remover um caracter (`del s[1]`)
- `str` é imutável

Strings com ' e "

Como escrever uma string que contém '?

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`
- `s = 'Olá "Mundo"'` funciona!

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`
- `s = 'Olá "Mundo"'` funciona!

Mas e se a string tiver tanto ' quanto "?

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`
- `s = 'Olá "Mundo"'` funciona!

Mas e se a string tiver tanto ' quanto "?

- `s = "I'm "nice" to people"` dá `SyntaxError`

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`
- `s = 'Olá "Mundo"'` funciona!

Mas e se a string tiver tanto ' quanto "?

- `s = "I'm "nice" to people"` dá `SyntaxError`
- `s = 'I'm "nice" to people'` dá `SyntaxError`

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`
- `s = 'Olá "Mundo"'` funciona!

Mas e se a string tiver tanto ' quanto "?

- `s = "I'm "nice" to people"` dá `SyntaxError`
- `s = 'I'm "nice" to people'` dá `SyntaxError`

Soluções:

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`
- `s = 'Olá "Mundo"'` funciona!

Mas e se a string tiver tanto ' quanto "?

- `s = "I'm "nice" to people"` dá `SyntaxError`
- `s = 'I'm "nice" to people'` dá `SyntaxError`

Soluções:

- `s = '''I'm "nice" to people'''` funciona

Strings com ' e "

Como escrever uma string que contém '?

- `s = 'I'm a coder'` dá `SyntaxError`
- `s = "I'm a coder"` funciona!

Como escrever uma string que contém "?

- `s = "Olá "Mundo""` dá `SyntaxError`
- `s = 'Olá "Mundo"'` funciona!

Mas e se a string tiver tanto ' quanto "?

- `s = "I'm "nice" to people"` dá `SyntaxError`
- `s = 'I'm "nice" to people'` dá `SyntaxError`

Soluções:

- `s = '''I'm "nice" to people'''` funciona
- `s = """I'm "nice" to people"""` funciona

Outra solução

Podemos escrever também:

Outra solução

Podemos escrever também:

- `'I\'m "nice" to people'` ou

Outra solução

Podemos escrever também:

- `'I\'m "nice" to people'` ou
- `"I'm \"nice\" to people"`

Outra solução

Podemos escrever também:

- `'I\'m "nice" to people'` ou
- `"I'm \"nice\" to people"`

Dizemos ao Python que:

Outra solução

Podemos escrever também:

- `'I\'m "nice" to people'` ou
- `"I'm \"nice\" to people"`

Dizemos ao Python que:

- ao invés de interpretar o `'` ou o `"` como final de string,

Outra solução

Podemos escrever também:

- `'I\'m "nice" to people'` ou
- `"I'm \"nice\" to people"`

Dizemos ao Python que:

- ao invés de interpretar o `'` ou o `"` como final de string,
- ele deve considerar como um caracter

Outra solução

Podemos escrever também:

- `'I\'m "nice" to people'` ou
- `"I'm \"nice\" to people"`

Dizemos ao Python que:

- ao invés de interpretar o `'` ou o `"` como final de string,
- ele deve considerar como um caracter

Estamos *escapando* o `'` ou o `"`

Outra solução

Podemos escrever também:

- `'I\'m "nice" to people'` ou
- `"I'm \"nice\" to people"`

Dizemos ao Python que:

- ao invés de interpretar o `'` ou o `"` como final de string,
- ele deve considerar como um caracter

Estamos *escapando* o `'` ou o `"`

- A `\` modifica a interpretação do símbolo a seguir

Escapando

Podemos usar a \ para:

Escapando

Podemos usar a \ para:

- Poder escrever ' e " em uma string

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

- `'\'` dá `SyntaxError`...

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

- `'\'` dá `SyntaxError`...
- `'\\'` é o correto

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

- `'\'` dá `SyntaxError`...
- `'\\'` é o correto

Lembre que `\`:

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

- `'\'` dá `SyntaxError`...
- `'\\'` é o correto

Lembre que `\`:

- Transforma caracteres especiais em caracteres normais

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

- `'\'` dá `SyntaxError`...
- `'\\'` é o correto

Lembre que `\`:

- Transforma caracteres especiais em caracteres normais
  - Ex: `'`, `"` e `\`

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

- `'\'` dá `SyntaxError`...
- `'\\'` é o correto

Lembre que `\`:

- Transforma caracteres especiais em caracteres normais
  - Ex: `'`, `"` e `\`
- E transforma caracteres normais em especiais

Escapando

Podemos usar a `\` para:

- Poder escrever `'` e `"` em uma string
- Inserir uma quebra de linha em um texto: `\n`
- Inserir um tab em um texto: `\t`

Mas e se eu quiser usar a `\` na minha string?

- `'\'` dá `SyntaxError`...
- `'\\'` é o correto

Lembre que `\`:

- Transforma caracteres especiais em caracteres normais
  - Ex: `'`, `"` e `\`
- E transforma caracteres normais em especiais
  - Ex: `\n` e `\t`

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \dots p_n$ uma palavra de n letras

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \ldots p_n$ uma palavra de n letras
  - p_1, p_2, ..., são as letras dessa palavra

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \ldots p_n$ uma palavra de n letras
  - p_1, p_2, ..., são as letras dessa palavra
- Seja $q = q_1 q_2 \ldots q_m$ uma palavra de m letras

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \ldots p_n$ uma palavra de n letras
  - p_1, p_2, ..., são as letras dessa palavra
- Seja $q = q_1 q_2 \ldots q_m$ uma palavra de m letras
- p precede q na ordem (escrevemos $p \prec q$) se:

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \ldots p_n$ uma palavra de n letras
  - p_1, p_2, ..., são as letras dessa palavra
- Seja $q = q_1 q_2 \ldots q_m$ uma palavra de m letras
- p precede q na ordem (escrevemos $p \prec q$) se:
  - p é prefixo próprio de q

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \dots p_n$ uma palavra de n letras
  - p_1, p_2, ..., são as letras dessa palavra
- Seja $q = q_1 q_2 \dots q_m$ uma palavra de m letras
- p precede q na ordem (escrevemos $p \prec q$) se:
  - p é prefixo próprio de q
  - Existe $1 \leq i \leq \min\{n, m\}$ tal que

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \dots p_n$ uma palavra de n letras
  - p_1, p_2, ..., são as letras dessa palavra
- Seja $q = q_1 q_2 \dots q_m$ uma palavra de m letras
- p precede q na ordem (escrevemos $p \prec q$) se:
  - p é prefixo próprio de q
  - Existe $1 \leq i \leq \min\{n, m\}$ tal que
    - $p_j = q_j$ para $0 \leq j < i$ e

Ordem lexicográfica (ou alfabética)

Na ordem alfabética (lexicográfica) temos que:

- **ana** vem antes de **beto**
- **abacate** vem antes de **ana**
- **ana** vem antes de **anamaria**

A ordem lexicográfica é definida da seguinte forma:

- Seja $p = p_1 p_2 \dots p_n$ uma palavra de n letras
  - p_1, p_2, ..., são as letras dessa palavra
- Seja $q = q_1 q_2 \dots q_m$ uma palavra de m letras
- p precede q na ordem (escrevemos $p \prec q$) se:
  - p é prefixo próprio de q
  - Existe $1 \leq i \leq \min\{n, m\}$ tal que
    - $p_j = q_j$ para $0 \leq j < i$ e
    - $p_i < q_i$

Exercício: menor lexicograficamente

Faça uma função que, dadas duas listas `l1` e `l2`, nos diz se `l1` é menor ou igual (lexicograficamente) a `l2`.

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!
- E você pode usar `<=`, `<`, `>=`, `>`, `==` e `!=`

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!
- E você pode usar `<=`, `<`, `>=`, `>`, `==` e `!=`

Exemplo:

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!
- E você pode usar `<=`, `<`, `>=`, `>`, `==` e `!=`

Exemplo:

- `"ana" < "beto"`

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!
- E você pode usar `<=`, `<`, `>=`, `>`, `==` e `!=`

Exemplo:

- `"ana" < "beto"`
- `"ana" < "anamaria"`

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!
- E você pode usar `<=`, `<`, `>=`, `>`, `==` e `!=`

Exemplo:

- `"ana" < "beto"`
- `"ana" < "anamaria"`
- `[1, 2, 3] < [1, 2, 4]`

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!
- E você pode usar `<=`, `<`, `>=`, `>`, `==` e `!=`

Exemplo:

- `"ana" < "beto"`
- `"ana" < "anamaria"`
- `[1, 2, 3] < [1, 2, 4]`
- `[1, 2, 3, 5] < [1, 2, 4]`

Comparação direta entre strings no Python

Na verdade, bastaria escrever `p <= q`!

- E isso vale também para strings!
- E você pode usar `<=`, `<`, `>=`, `>`, `==` e `!=`

Exemplo:

- `"ana" < "beto"`
- `"ana" < "anamaria"`
- `[1, 2, 3] < [1, 2, 4]`
- `[1, 2, 3, 5] < [1, 2, 4]`
- `[1, 2] < [1, 2, 4]`

Exercícios

1. Faça uma função que, dado uma string `sep` e uma lista `l` de strings, concatena as strings de `l` usando `sep` como separador
   - Ex: Se `l == ["A", "B", "C"]` e `sep == ","`, então o resultado deve ser `"A,B,C"`

Exercícios

1. Faça uma função que, dado uma string `sep` e uma lista `l` de strings, concatena as strings de `l` usando `sep` como separador
   - Ex: Se `l == ["A", "B", "C"]` e `sep == ","`, então o resultado deve ser `"A,B,C"`
2. Faça uma função que, dadas duas strings `s` e `t`, verifica se `s` é prefixo de `t`

Exercícios

1. Faça uma função que, dado uma string `sep` e uma lista `l` de strings, concatena as strings de `l` usando `sep` como separador
   - Ex: Se `l == ["A", "B", "C"]` e `sep == ","`, então o resultado deve ser `"A,B,C"`
2. Faça uma função que, dadas duas strings `s` e `t`, verifica se `s` é prefixo de `t`
3. Faça uma função que, dadas duas strings `s` e `t`, verifica se `s` é substring de `t`

Exercícios

1. Faça uma função que, dado uma string `sep` e uma lista `l` de strings, concatena as strings de `l` usando `sep` como separador
   - Ex: Se `l == ["A", "B", "C"]` e `sep == ","`, então o resultado deve ser `"A,B,C"`
2. Faça uma função que, dadas duas strings `s` e `t`, verifica se `s` é prefixo de `t`
3. Faça uma função que, dadas duas strings `s` e `t`, verifica se `s` é substring de `t`
4. Faça uma função que, dada uma string `s` e uma string `sep`, devolve uma lista resultante da quebra de `s` em uma ou mais strings, em cada ocorrência de `sep`
   - Ex: Se `s == "A,B,C"` e `sep == ","`, então o resultado deve ser `["A", "B", "C"]`

Alguns métodos úteis

`sep.join(lista)`:

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

`s.split(sep)`:

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

`s.split(sep)`:

- Quebra string `s` em cada ocorrência de `sep` em uma lista

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

`s.split(sep)`:

- Quebra string `s` em cada ocorrência de `sep` em uma lista
- Ex: `"A,B,C".split(",")` resulta em `"A,B,C"`

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

`s.split(sep)`:

- Quebra string `s` em cada ocorrência de `sep` em uma lista
- Ex: `"A,B,C".split(",")` resulta em `"A,B,C"`
- Ex: `palavras = input().split(' ')`

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

`s.split(sep)`:

- Quebra string `s` em cada ocorrência de `sep` em uma lista
- Ex: `"A,B,C".split(",")` resulta em `"A,B,C"`
- Ex: `palavras = input().split(' ')`

Outros métodos que fazem o mesmo que fizemos nos exercícios:

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

`s.split(sep)`:

- Quebra string `s` em cada ocorrência de `sep` em uma lista
- Ex: `"A,B,C".split(",")` resulta em `"A,B,C"`
- Ex: `palavras = input().split(' ')`

Outros métodos que fazem o mesmo que fizemos nos exercícios:

- `s.startswith` para verificar se uma string é substring no começo (ou meio) de `s`

Alguns métodos úteis

`sep.join(lista)`:

- Concatena uma `lista` usando `sep` como separador
- Ex: `",".join(["A", "B", "C"])` resulta em `"A,B,C"`
- Ex: `"".join(["A", "B", "C"])` resulta em `"ABC"`

`s.split(sep)`:

- Quebra string `s` em cada ocorrência de `sep` em uma lista
- Ex: `"A,B,C".split(",")` resulta em `"A,B,C"`
- Ex: `palavras = input().split(' ')`

Outros métodos que fazem o mesmo que fizemos nos exercícios:

- `s.startswith` para verificar se uma string é substring no começo (ou meio) de `s`
- `s.find` para achar o índice onde uma substring começa em `s`

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

```
s = f'{x}**{y} é {x ** y}'
```

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

```
s = f'{x}**{y} é {x ** y}'
```

- O `f` indica que essa é uma string formatada

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

```
s = f'{x}**{y} é {x ** y}'
```

- O `f` indica que essa é uma string formatada
- Entre `{}` podemos colocar qualquer expressão Python

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

```
s = f'{x}**{y} é {x ** y}'
```

- O `f` indica que essa é uma string formatada
- Entre `{}` podemos colocar qualquer expressão Python
- Será convertida para `str` automaticamente

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

```
s = f'{x}**{y} é {x ** y}'
```

- O `f` indica que essa é uma string formatada
- Entre `{}` podemos colocar qualquer expressão Python
- Será convertida para `str` automaticamente

Podemos também escrever:

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

```
s = f'{x}**{y} é {x ** y}'
```

- O `f` indica que essa é uma string formatada
- Entre `{}` podemos colocar qualquer expressão Python
- Será convertida para `str` automaticamente

Podemos também escrever:

```
s = '{}**{} é {}'.format(x, y, x ** y)
```

Formatação de Strings

Queremos montar uma string a partir de cálculos que fizemos:

```
s = str(x) + '**' + str(y) + "é "+ str(x ** y)
```

- Temos que converter para `str` para concatenar...
- O que deixa a expressão longa

Podemos escrever, simplesmente:

```
s = f'{x}**{y} é {x ** y}'
```

- O `f` indica que essa é uma string formatada
- Entre `{}` podemos colocar qualquer expressão Python
- Será convertida para `str` automaticamente

Podemos também escrever:

```
s = '{}**{} é {}'.format(x, y, x ** y)
```

- `format` troca o `i`-ésimo `{}` pelo `i`-ésimo parâmetro

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`
  - `{i}` é trocado pelo `i`-ésimo parâmetro

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`
  - `{i}` é trocado pelo `i`-ésimo parâmetro
  - Não precisa ser na mesma ordem...

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`
  - `{i}` é trocado pelo `i`-ésimo parâmetro
  - Não precisa ser na mesma ordem...
  - Ex: `'{1} {0} {1}'.format('A', 'B') == 'B A B'`

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`
  - `{i}` é trocado pelo `i`-ésimo parâmetro
  - Não precisa ser na mesma ordem...
  - Ex: `'{1} {0} {1}'.format('A', 'B') == 'B A B'`
- Opção 2: `'{base}**{exp} é {res}'.format(base=x, exp=y, res=x ** y)`

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`
  - `{i}` é trocado pelo `i`-ésimo parâmetro
  - Não precisa ser na mesma ordem...
  - Ex: `'{1} {0} {1}'.format('A', 'B') == 'B A B'`
- Opção 2: `'{base}**{exp} é {res}'.format(base=x, exp=y, res=x ** y)`
  - Usa a ideia de nome de parâmetro

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`
  - `{i}` é trocado pelo `i`-ésimo parâmetro
  - Não precisa ser na mesma ordem...
  - Ex: `'{1} {0} {1}'.format('A', 'B') == 'B A B'`
- Opção 2: `'{base}**{exp} é {res}'.format(base=x, exp=y, res=x ** y)`
  - Usa a ideia de nome de parâmetro
- Opção 3: Combinar os dois...o que talvez seja meio estranho `'{0}**{exp} é {res}'.format(x, exp=y, res=x ** y)`

`format` mais legível

`'{}**{} é {}'.format(x, y, x ** y)` não é tão legível...

- Opção 1: `'{0}**{1} é {2}'.format(x, y, x ** y)`
  - `{i}` é trocado pelo `i`-ésimo parâmetro
  - Não precisa ser na mesma ordem...
  - Ex: `'{1} {0} {1}'.format('A', 'B') == 'B A B'`
- Opção 2: `'{base}**{exp} é {res}'.format(base=x, exp=y, res=x ** y)`
  - Usa a ideia de nome de parâmetro
- Opção 3: Combinar os dois...o que talvez seja meio estranho `'{0}**{exp} é {res}'.format(x, exp=y, res=x ** y)`
  - Não vejo motivo para usar esse...

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`
  - E o `.1` indica que queremos uma casa de precisão

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`
  - E o `.1` indica que queremos uma casa de precisão
  - Ex: `'{0:.2f}'.format(x)` é `'0.30'`

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`
  - E o `.1` indica que queremos uma casa de precisão
  - Ex: `'{0:.2f}'.format(x)` é `'0.30'`
  - Ex: `'{:.0f}'.format(x)` é `'0'`

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`
  - E o `.1` indica que queremos uma casa de precisão
  - Ex: `'{0:.2f}'.format(x)` é `'0.30'`
  - Ex: `'{:.0f}'.format(x)` é `'0'`
- `'{:e}'.format(x)` é `'3.000000e-01'`

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`
  - E o `.1` indica que queremos uma casa de precisão
  - Ex: `'{0:.2f}'.format(x)` é `'0.30'`
  - Ex: `'{:.0f}'.format(x)` é `'0'`
- `'{:e}'.format(x)` é `'3.000000e-01'`
  - `e` indica que queremos notação científica

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`
  - E o `.1` indica que queremos uma casa de precisão
  - Ex: `'{0:.2f}'.format(x)` é `'0.30'`
  - Ex: `'{:.0f}'.format(x)` é `'0'`
- `'{:e}'.format(x)` é `'3.000000e-01'`
  - `e` indica que queremos notação científica
  - Com `E` maiúsculo, fica `'3.000000E-01'`

Formatação dos dados

Imagine que `x = 0.1 + 0.2` e queremos colocar `x` em uma string

- `'{}'.format(x)` é `'0.30000000000000004'`
- `'{num:.1f}'.format(num=x)` é `'0.3'`
  - O `f` indica que queremos imprimir um `float`
  - E o `.1` indica que queremos uma casa de precisão
  - Ex: `'{0:.2f}'.format(x)` é `'0.30'`
  - Ex: `'{:.0f}'.format(x)` é `'0'`
- `'{:e}'.format(x)` é `'3.000000e-01'`
  - `e` indica que queremos notação científica
  - Com `E` maiúsculo, fica `'3.000000E-01'`
- `'{:%}'.format(x)` é `'30.000000%'`

Formatação dos dados

Temos opções para `int` também

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)
- `'{:b}'.format(42)` é `'101010'` (binário)

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)
- `'{:b}'.format(42)` é `'101010'` (binário)
- E com um `#` antes de `x`, `o`, e `b` você obtém:

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)
- `'{:b}'.format(42)` é `'101010'` (binário)
- E com um `#` antes de `x`, `o`, e `b` você obtém:
  - `'0x2a'`, `'0o52'`, e `'0b101010'`

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)
- `'{:b}'.format(42)` é `'101010'` (binário)
- E com um `#` antes de `x`, `o`, e `b` você obtém:
  - `'0x2a'`, `'0o52'`, e `'0b101010'`

Há tem algumas opções (`<`, `>` e `^`) para controlar o tamanho da string e alinhar o conteúdo

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)
- `'{:b}'.format(42)` é `'101010'` (binário)
- E com um `#` antes de `x`, `o`, e `b` você obtém:
  - `'0x2a'`, `'0o52'`, e `'0b101010'`

Há tem algumas opções (`<`, `>` e `^`) para controlar o tamanho da string e alinhar o conteúdo

- Além de mais algumas opções de impressão de número

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)
- `'{:b}'.format(42)` é `'101010'` (binário)
- E com um `#` antes de `x`, `o`, e `b` você obtém:
  - `'0x2a'`, `'0o52'`, e `'0b101010'`

Há tem algumas opções (`<`, `>` e `^`) para controlar o tamanho da string e alinhar o conteúdo

- Além de mais algumas opções de impressão de número
  - Ex: Em números positivos, o sinal deve aparecer ou não?

Formatação dos dados

Temos opções para `int` também

- `'{:d}'.format(42)` é `'42'` (decimal)
- `'{:x}'.format(42)` é `'2a'` (hexadecimal)
- `'{:X}'.format(42)` é `'2A'` (hexadecimal)
- `'{:o}'.format(42)` é `'52'` (octal)
- `'{:b}'.format(42)` é `'101010'` (binário)
- E com um `#` antes de `x`, `o`, e `b` você obtém:
  - `'0x2a'`, `'0o52'`, e `'0b101010'`

Há tem algumas opções (`<`, `>` e `^`) para controlar o tamanho da string e alinhar o conteúdo

- Além de mais algumas opções de impressão de número
  - Ex: Em números positivos, o sinal deve aparecer ou não?
- Pesquise sobre isso!

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

- ao invés de escrever `print(f'x = {x}')`

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

- ao invés de escrever `print(f'x = {x}')`
- posso escrever `print(f'{x = }')`

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

- ao invés de escrever `print(f'x = {x}')`
- posso escrever `print(f'{x = }')`
- na verdade, o `print` não importa para a string

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

- ao invés de escrever `print(f'x = {x}')`
- posso escrever `print(f'{x = }')`
- na verdade, o `print` não importa para a string

Outro exemplo (com `x = 10` e `y = 3`):

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

- ao invés de escrever `print(f'x = {x}')`
- posso escrever `print(f'{x = }')`
- na verdade, o `print` não importa para a string

Outro exemplo (com `x = 10` e `y = 3`):
`f'x + 3 * y + 2 = '` é a string `'x + 3 * y + 2 = 21'`

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

- ao invés de escrever `print(f'x = {x}')`
- posso escrever `print(f'{x = }')`
- na verdade, o `print` não importa para a string

Outro exemplo (com `x = 10` e `y = 3`):
`f'x + 3 * y + 2 = '` é a string `'x + 3 * y + 2 = 21'`

Isso não é muito bonito para o usuário

Uma última dica

Muitas vezes queremos imprimir o valor de uma variável

- ao invés de escrever `print(f'x = {x}')`
- posso escrever `print(f'{x = }')`
- na verdade, o `print` não importa para a string

Outro exemplo (com `x = 10` e `y = 3`):
`f'x + 3 * y + 2 = '` é a string `'x + 3 * y + 2 = 21'`

Isso não é muito bonito para o usuário

- mas pode ser útil para o programador

`str` tem muitos métodos úteis

`capitalize` • `casefold` • `center` • `count` • `encode` • `endswith` • `expandtabs` • `find` • `format_map` • `format` • `index` • `isalnum` • `isalpha` • `isascii` • `isdecimal` • `isdigit` • `isidentifier` • `islower` • `isnumeric` • `isprintable` • `isspace` • `istitle` • `isupper` • `join` • `ljust` • `lower` • `lstrip` • `partition` • `replace` • `rfind` • `rindex` • `rjust` • `rpartition` • `rsplit` • `rstrip` • `split` • `splitlines` • `startswith` • `strip` • `swapcase` • `title` • `translate` • `upper` • `zfill`

Leia a documentação!