

Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:~<10}  
número: tamanho 5 com zeros    {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `FK="BAKFHWPOX"`
Efetue todos os fatiamentos.

- FK[:9:2]
- FK[-8:-3:2]
- FK[-9:9:2]
- FK[:7:-1]
- FK[:9:2]

2. Considere `BX="JLYKOVGP"`
Efetue todos os fatiamentos.

- BX[3:8:2]
- BX[3:6:2]
- BX[-8:8]
- BX[3:6:3]
- BX[2:8:2]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn't'
"doesn't"
>>> "doesn't"
"doesn't"
>>> "Yes," he said.'
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")
>>> L[0]='X'
>>> print(L)
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']
>>> s=''.join(L)
>>> s
'Xuritiba'
>>> print(s)
Xuritiba
```

Como se viu a função list transforma cada caractere da string em um elemento da lista retornada. O método join faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como '\t', '\xc3\xa9' etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando print produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\ndiversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]
'a'
>>> palavra[0:2]
'Aj'
>>> palavra[2:4]
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'
=== Erro ===
>>> palavra[:1] = 'Splat'
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]
'xjudaZ'
>>> 'Splat' + palavra[5]
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: s[:i] + s[i] é igual a s.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]
'judaz'
>>> palavra[10:]
''
```

```
>>> palavra[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que -0 é o mesmo que 0, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]
'A'
```

A função built-in (embutida) len() devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'
>>> len(s)
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()
'Curitiba'
>>> "ágora".capitalize()
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)
'      paraná      '
>>> "paraná".center(2)
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna -1.

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam True caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']
>>> ''.join(a)
'aguafogoxabu'
>>> '\n'.join(a)
'agua\nfogo\nxabu'
>>> print('\n'.join(a))
agua
fogo
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')
('curitibapa', 'ra', 'nabrasil')
>>> "curitibaparanabrasil".partition('z')
('curitibaparanabrasil', '', '')
```

.replace() Substitui n ocorrências da substring antiga pela nova. Se n não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")
'curitibapaagoranabagorasil'
>>> "curitibaparanabrasil".replace("ra", "zz", 1)
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador n vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')
['curitib', 'p', 'r', 'n', 'br', 'sil']
>>> "curitiba parana brasil".split()
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com %d, %s, %f, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}
alinhar à esquerda              {0:<10}
centralizar no espaço           {0:^10}
preencher o espaço com .       {0:.<10}
número: tamanho 5 com zeros     {0:05}
idem: 3 casas após a vírgula    {0:10.3}
com + ou -                      {0:~10}{1:+3}
em binário                      {0:b}
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo unicode. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'
u'Hello World !'
```

O u antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

O código de escape \u0020 insere um caractere Unicode com valor ordinal 0x0020 (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere SM="IZTEKSOQ" Efetue todos os fatiamentos.

- a) SM[3:8:2]
- b) SM[:6]
- c) SM[-7:6:2]
- d) SM[-8:8:3]
- e) SM[-8:7:2]

2. Considere RC="GFPJQZND" Efetue todos os fatiamentos.

- a) RC[:8:2]
- b) RC[2:-2:2]
- c) RC[2:8:2]
- d) RC[2:6:2]
- e) RC[:7]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:~<10}  
número: tamanho 5 com zeros     {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `MF="PHGQXJZ"`
Efetue todos os fatiamentos.

- `MF[-6:5]`
- `MF[2:5:2]`
- `MF[-5:-3]`
- `MF[:7:3]`
- `MF[:5]`

2. Considere `CK="LIAKEPZMV"`
Efetue todos os fatiamentos.

- `CK[-7:-3]`
- `CK[-9:8]`
- `CK[:7:-2]`
- `CK[3:8:3]`
- `CK[2:-1:2]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho          {0:10}  
alinhar à esquerda             {0:<10}  
centralizar no espaço          {0:^10}  
preencher o espaço com .      {0:.<10}  
número: tamanho 5 com zeros   {0:05}  
idem: 3 casas após a vírgula  {0:10.3}  
com + ou -                     {0:~10}{1:+3}  
em binário                     {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `ZW="TXUSBI"`
Efetue todos os fatiamentos.

- `ZW[2:-3:2]`
- `ZW[4:-2]`
- `ZW[3:6]`
- `ZW[-4:6:2]`
- `ZW[5:3]`

2. Considere `JN="LDNBHW"`
Efetue todos os fatiamentos.

- `JN[3:4:2]`
- `JN[:4:3]`
- `JN[3:4:3]`
- `JN[:6:3]`
- `JN[3:6]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessadas como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros     {0:05}  
idem: 3 casas após a vírgula    {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existent, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `EM="KDIQJC"`
Efetue todos os fatiamentos.

- `EM[2:-3:3]`
- `EM[:5:2]`
- `EM[1:6:2]`
- `EM[:4:3]`
- `EM[3:6:2]`

2. Considere `CQ="JRGKATC"`
Efetue todos os fatiamentos.

- `CQ[2:6:2]`
- `CQ[2:-2]`
- `CQ[:5:-2]`
- `CQ[-6:5:2]`
- `CQ[2:5:2]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn't'
"doesn't"
>>> "doesn't"
"doesn't"
>>> "Yes," he said.'
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")
>>> L[0]='X'
>>> print(L)
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']
>>> s=''.join(L)
>>> s
'Xuritiba'
>>> print(s)
Xuritiba
```

Como se viu a função list transforma cada caractere da string em um elemento da lista retornada. O método join faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como '\t', '\xc3\xa9' etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando print produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\ndiversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]
'a'
>>> palavra[0:2]
'Aj'
>>> palavra[2:4]
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'
=== Erro ===
>>> palavra[:1] = 'Splat'
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]
'xjudaZ'
>>> 'Splat' + palavra[5]
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: s[:i] + s[i] é igual a s.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]
'judaz'
>>> palavra[10:]
''
```

```
>>> palavra[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que -0 é o mesmo que 0, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]
'A'
```

A função built-in (embutida) len() devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'
>>> len(s)
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()
'Curitiba'
>>> "ágora".capitalize()
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)
'   paraná   '
>>> "paraná".center(2)
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna -1.

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

Avaliadores São eles: .isalpha(), .isdecimal(), .isdigit(), .islower(), .isnumeric(), .isspace(), .istitle(), .isupper(). Retornam True caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']
>>> ''.join(a)
'aguafogoxabu'
>>> '\n'.join(a)
'agua\nfogo\nxabu'
>>> print('\n'.join(a))
agua
fogo
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')
('curitibapa', 'ra', 'nabrasil')
>>> "curitibaparanabrasil".partition('z')
('curitibaparanabrasil', '', '')
```

.replace() Substitui n ocorrências da substring antiga pela nova. Se n não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")
'curitibapaagoranabagorasil'
>>> "curitibaparanabrasil".replace("ra", "zz", 1)
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador n vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')
['curitib', 'p', 'r', 'n', 'br', 'sil']
>>> "curitiba parana brasil".split()
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com %d, %s, %f, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}
alinhar à esquerda              {0:<10}
centralizar no espaço           {0:^10}
preencher o espaço com .       {0:~<10}
número: tamanho 5 com zeros     {0:05}
idem: 3 casas após a vírgula    {0:10.3}
com + ou -                      {0:-10}{1:+3}
em binário                      {0:b}
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo unicode. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja http://www.unicode.org/) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'
u'Hello World !'
```

O u antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

O código de escape \u0020 insere um caractere Unicode com valor ordinal 0x0020 (o espaço em branco) naquela posição.

Para você fazer

1. Considere XE="NPATZBFO" Efetue todos os fatiamentos.

- a) XE[-6:8:2]
b) XE[1:-2]
c) XE[-6:-2]
d) XE[3:6:3]
e) XE[2:-1:3]

2. Considere SP="HBJTAWLCS" Efetue todos os fatiamentos.

- a) SP[-9:9]
b) SP[-7:-1:2]
c) SP[3:8:2]
d) SP[3:9:3]
e) SP[-7:-2:3]

Responda aqui:

Table with 5 columns (1a-1e, 2a-2e) and 2 rows.



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessadas como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros    {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `TQ="JPUEHVNRW"`
Efetue todos os fatiamentos.

- TQ[2:8:2]
- TQ[2:7:2]
- TQ[-7:8:2]
- TQ[2:9]
- TQ[2:-2:2]

2. Considere `QD="WMBJZEI"`
Efetue todos os fatiamentos.

- QD[1:7:2]
- QD[1:-1:2]
- QD[-6:-1:2]
- QD[3:6]
- QD[6:2]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho          {0:10}  
alinhar à esquerda            {0:<10}  
centralizar no espaço         {0:^10}  
preencher o espaço com .     {0:~<10}  
número: tamanho 5 com zeros  {0:05}  
idem: 3 casas após a vírgula {0:10.3}  
com + ou -                    {0:-10}{1:+3}  
em binário                    {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

📖 Para você fazer

1. Considere `AE="FUJOKNZR"`
Efetue todos os fatiamentos.

- `AE[3:-1:2]`
- `AE[1:7:2]`
- `AE[:6:2]`
- `AE[-7:-2:3]`
- `AE[:6:-2]`

2. Considere `MH="CSWHZB"`
Efetue todos os fatiamentos.

- `MH[3:4:3]`
- `MH[2:5:2]`
- `MH[-4:-1:2]`
- `MH[2:5:2]`
- `MH[2:6:2]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador `n` vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros     {0:05}  
idem: 3 casas após a vírgula    {0:10.3}  
com + ou -                       {0:~10}{1:+3}  
em binário                       {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `PQ="DBUIVZAF"`
Efetue todos os fatiamentos.

- `PQ[-7:7:2]`
- `PQ[-9:-1:2]`
- `PQ[2:8:3]`
- `PQ[-9:-3:2]`
- `PQ[1:-2:3]`

2. Considere `TS="CAJMQEG"`
Efetue todos os fatiamentos.

- `TS[1:-1]`
- `TS[-6:5:2]`
- `TS[3:7:2]`
- `TS[:5:3]`
- `TS[2:-1:2]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessadas como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:~<10}  
número: tamanho 5 com zeros    {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

📖 Para você fazer

1. Considere `BY="IPFYMU"`
Efetue todos os fatiamentos.

- BY[:5]
- BY[1:5:2]
- BY[:6]
- BY[-6:-3:2]
- BY[3:6:2]

2. Considere `PC="YJDAPVXRU"`
Efetue todos os fatiamentos.

- PC[2:-1:2]
- PC[-7:9:3]
- PC[:8:2]
- PC[:7:-1]
- PC[2:7:3]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho          {0:10}  
alinhar à esquerda             {0:<10}  
centralizar no espaço          {0:^10}  
preencher o espaço com .      {0:~<10}  
número: tamanho 5 com zeros   {0:05}  
idem: 3 casas após a vírgula  {0:10.3}  
com + ou -                     {0:~10}{1:+3}  
em binário                     {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existent, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `TD="GYRJLCU"`
Efetue todos os fatiamentos.

- TD[-5:-2]
- TD[:6]
- TD[3:7:2]
- TD[2:5:2]
- TD[:5:-3]

2. Considere `KC="OUPJZF"`
Efetue todos os fatiamentos.

- KC[:5:2]
- KC[1:5:2]
- KC[-6:6:3]
- KC[3:-1:2]
- KC[2:6:3]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessadas como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador `n` vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros    {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `BX="RGOQYPW"`
Efetue todos os fatiamentos.
 - a) `BX[3:7:3]`
 - b) `BX[1:-1]`
 - c) `BX[:6:2]`
 - d) `BX[-7:7:2]`
 - e) `BX[2:6]`

2. Considere `DP="ISQRCDVZ"`
Efetue todos os fatiamentos.
 - a) `DP[:6:2]`
 - b) `DP[:8:2]`
 - c) `DP[2:7:3]`
 - d) `DP[-7:8:2]`
 - e) `DP[1:6]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros    {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

📖 Para você fazer

1. Considere `DE="JAUSOKN"`
Efetue todos os fatiamentos.

- DE[-7:6:2]
- DE[-6:7]
- DE[1:6:2]
- DE[:5]
- DE[2:-2:2]

2. Considere `EF="HDPGQYM"`
Efetue todos os fatiamentos.

- EF[-7:6:2]
- EF[-5:5:2]
- EF[-5:-2]
- EF[:6]
- EF[1:-2:3]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessadas como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i:]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros     {0:05}  
idem: 3 casas após a vírgula    {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existent, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

📖 Para você fazer

1. Considere `AF="NAMXIWD"`
Efetue todos os fatiamentos.

- `AF[3:7:2]`
- `AF[3:5:2]`
- `AF[:5:2]`
- `AF[3:5]`
- `AF[-7:6:2]`

2. Considere `CB="UYSWNHA"`
Efetue todos os fatiamentos.

- `CB[:5:-1]`
- `CB[3:6:2]`
- `CB[:7:2]`
- `CB[:5:3]`
- `CB[3:7]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn't'
"doesn't"
>>> "doesn't"
"doesn't"
>>> "Yes," he said.'
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")
>>> L[0]='X'
>>> print(L)
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']
>>> s=''.join(L)
>>> s
'Xuritiba'
>>> print(s)
Xuritiba
```

Como se viu a função list transforma cada caractere da string em um elemento da lista retornada. O método join faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como '\t', '\xc3\xa9' etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando print produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\ndiversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]
'a'
>>> palavra[0:2]
'Aj'
>>> palavra[2:4]
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'
=== Erro ===
>>> palavra[:1] = 'Splat'
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]
'xjudaZ'
>>> 'Splat' + palavra[5]
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: s[:i] + s[i] é igual a s.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]
'judaz'
>>> palavra[10:]
''
```

```
>>> palavra[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que -0 é o mesmo que 0, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]
'A'
```

A função built-in (embutida) len() devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'
>>> len(s)
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()
'Curitiba'
>>> "ágora".capitalize()
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)
'   paraná   '
>>> "paraná".center(2)
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna -1.

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

Avaliadores São eles: .isalpha(), .isdecimal(), .isdigit(), .islower(), .isnumeric(), .isspace(), .istitle(), .isupper(). Retornam True caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']
>>> ''.join(a)
'aguafogo xabu'
>>> '\n'.join(a)
'agua\nfogo\nxabu'
>>> print('\n'.join(a))
agua
fogo
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')
('curitibapa', 'ra', 'nabrasil')
>>> "curitibaparanabrasil".partition('z')
('curitibaparanabrasil', '', '')
```

.replace() Substitui n ocorrências da substring antiga pela nova. Se n não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")
'curitibapaagoranabagorasil'
>>> "curitibaparanabrasil".replace("ra", "zz", 1)
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador n vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')
['curitib', 'p', 'r', 'n', 'br', 'sil']
>>> "curitiba parana brasil".split()
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com %d, %s, %f, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}
alinhar à esquerda              {0:<10}
centralizar no espaço           {0:^10}
preencher o espaço com .       {0:~<10}
número: tamanho 5 com zeros     {0:05}
idem: 3 casas após a vírgula    {0:10.3}
com + ou -                      {0:~10}{1:+3}
em binário                      {0:b}
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo unicode. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja http://www.unicode.org/) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'
u'Hello World !'
```

O u antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

O código de escape \u0020 insere um caractere Unicode com valor ordinal 0x0020 (o espaço em branco) naquela posição.

Para você fazer

- 1. Considere IO="KYUFNVGBW"
Efetue todos os fatiamentos.
a) IO[2:9:2]
b) IO[3:7:3]
c) IO[3:-2]
d) IO[:7:-2]
e) IO[2:8]

- 2. Considere KM="DLAHQYC"
Efetue todos os fatiamentos.
a) KM[:7:2]
b) KM[3:6]
c) KM[2:7:2]
d) KM[:7]
e) KM[-7:-3:2]

Responda aqui:

Table with 5 columns (1a-1e, 2a-2e) and 2 rows.



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessadas como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador `n` vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho      {0:10}  
alinhar à esquerda         {0:<10}  
centralizar no espaço      {0:^10}  
preencher o espaço com .  {0:~<10}  
número: tamanho 5 com zeros {0:05}  
idem: 3 casas após a vírgula {0:10.3}  
com + ou -                  {0:~10}{1:+3}  
em binário                  {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `PL="JWSGCLQZ"`
Efetue todos os fatiamentos.
 - a) `PL[-6:7:2]`
 - b) `PL[-6:8:2]`
 - c) `PL[:7:3]`
 - d) `PL[1:8:2]`
 - e) `PL[3:7:2]`

2. Considere `OC="JBMPQX"`
Efetue todos os fatiamentos.
 - a) `OC[1:5:2]`
 - b) `OC[3:4]`
 - c) `OC[:6]`
 - d) `OC[:6:3]`
 - e) `OC[:4:3]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessadas como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros    {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

📖 Para você fazer

1. Considere `AE="FISBGA"`
Efetue todos os fatiamentos.

- AE[-5:5]
- AE[2:5]
- AE[:6]
- AE[3:-2]
- AE[:4:2]

2. Considere `GY="NMHWSIXA"`
Efetue todos os fatiamentos.

- GY[:8:2]
- GY[3:8:2]
- GY[3:6:2]
- GY[3:-3:2]
- GY[1:7]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn't'
"doesn't"
>>> "doesn't"
"doesn't"
>>> "Yes," he said.'
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")
>>> L[0]='X'
>>> print(L)
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']
>>> s=''.join(L)
>>> s
'Xuritiba'
>>> print(s)
Xuritiba
```

Como se viu a função list transforma cada caractere da string em um elemento da lista retornada. O método join faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como '\t', '\xc3\xa9' etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando print produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\ndiversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]
'a'
>>> palavra[0:2]
'Aj'
>>> palavra[2:4]
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'
=== Erro ===
>>> palavra[:1] = 'Splat'
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]
'xjudaZ'
>>> 'Splat' + palavra[5]
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: s[:i] + s[i] é igual a s.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]
'judaZ'
>>> palavra[10:]
''
```

```
>>> palavra[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que -0 é o mesmo que 0, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]
'A'
```

A função built-in (embutida) len() devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'
>>> len(s)
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()
'Curitiba'
>>> "ágora".capitalize()
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)
'   paraná   '
>>> "paraná".center(2)
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna -1.

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

Avaliadores São eles: .isalpha(), .isdecimal(), .isdigit(), .islower(), .isnumeric(), .isspace(), .istitle(), .isupper(). Retornam True caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']
>>> ''.join(a)
'aguafogo xabu'
>>> '\n'.join(a)
'agua\nfogo\nxabu'
>>> print('\n'.join(a))
agua
fogo
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')
('curitibapa', 'ra', 'nabrasil')
>>> "curitibaparanabrasil".partition('z')
('curitibaparanabrasil', '', '')
```

.replace() Substitui n ocorrências da substring antiga pela nova. Se n não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")
'curitibapaagoranabagorasil'
>>> "curitibaparanabrasil".replace("ra", "zz", 1)
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador n vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')
['curitib', 'p', 'r', 'n', 'br', 'sil']
>>> "curitiba parana brasil".split()
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com %d, %s, %f, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}
alinhar à esquerda              {0:<10}
centralizar no espaço           {0:^10}
preencher o espaço com .       {0:~<10}
número: tamanho 5 com zeros    {0:05}
idem: 3 casas após a vírgula   {0:10.3}
com + ou -                      {0:~10}{1:+3}
em binário                      {0:b}
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo unicode. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'
u'Hello World !'
```

O u antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

O código de escape \u0020 insere um caractere Unicode com valor ordinal 0x0020 (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere EB="BCZLRPXSG" Efetue todos os fatiamentos.

- a) EB[:7:-3]
- b) EB[3:8:2]
- c) EB[-9:9:3]
- d) EB[-9:7]
- e) EB[2:8:3]

2. Considere KN="RYKCOIB" Efetue todos os fatiamentos.

- a) KN[2:5]
- b) KN[-5:-2:2]
- c) KN[3:5]
- d) KN[2:5:2]
- e) KN[1:5:2]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador `n` vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros    {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:-10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `IM="GUPFONCR"`
Efetue todos os fatiamentos.
 - a) `IM[:8:3]`
 - b) `IM[-7:6]`
 - c) `IM[:6:2]`
 - d) `IM[-6:-2]`
 - e) `IM[-6:8:2]`

2. Considere `MI="ADQHBF"`
Efetue todos os fatiamentos.
 - a) `MI[1:4:3]`
 - b) `MI[:6:2]`
 - c) `MI[:5:2]`
 - d) `MI[2:-2:2]`
 - e) `MI[:5:3]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho      {0:10}  
alinhar à esquerda         {0:<10}  
centralizar no espaço      {0:^10}  
preencher o espaço com .  {0:.<10}  
número: tamanho 5 com zeros {0:05}  
idem: 3 casas após a vírgula {0:10.3}  
com + ou -                 {0:-10}{1:+3}  
em binário                 {0:b}  
em hexadecimal*           {0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

📖 Para você fazer

1. Considere `OW="YGSILJO"`
Efetue todos os fatiamentos.

- `OW[:5:-1]`
- `OW[-7:6]`
- `OW[:7]`
- `OW[-7:-3:2]`
- `OW[-5:-2:2]`

2. Considere `FQ="GUBANOFICI"`
Efetue todos os fatiamentos.

- `FQ[3:8]`
- `FQ[:7:-1]`
- `FQ[-7:-2:2]`
- `FQ[3:7:2]`
- `FQ[-9:-3:2]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i:]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador `n` vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho      {0:10}  
alinhar à esquerda         {0:<10}  
centralizar no espaço      {0:^10}  
preencher o espaço com .  {0:.<10}  
número: tamanho 5 com zeros {0:05}  
idem: 3 casas após a vírgula {0:10.3}  
com + ou -                 {0:~10}{1:+3}  
em binário                 {0:b}  
em hexadecimal*           {0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

- Considere `AH="AMSTNWBF"`
Efetue todos os fatiamentos.
 - AH[:6:-2]
 - AH[1:8]
 - AH[:6]
 - AH[2:-3]
 - AH[2:6:2]

- Considere `KZ="CQNMMSW"`
Efetue todos os fatiamentos.
 - KZ[1:-3:2]
 - KZ[:4:3]
 - KZ[1:5:2]
 - KZ[3:5:2]
 - KZ[:6:3]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i:]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judazZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador `n` vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho      {0:10}  
alinhar à esquerda         {0:<10}  
centralizar no espaço      {0:^10}  
preencher o espaço com .   {0:~<10}  
número: tamanho 5 com zeros {0:05}  
idem: 3 casas após a vírgula {0:10.3}  
com + ou -                  {0:~10}{1:+3}  
em binário                  {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `BC="VEPTMWF"`
Efetue todos os fatiamentos.

- BC[3:5]
- BC[3:6:2]
- BC[1:-3:2]
- BC[-7:-2]
- BC[:6:2]

2. Considere `TJ="NZVUHS"`
Efetue todos os fatiamentos.

- TJ[3:6]
- TJ[:6:3]
- TJ[1:6:2]
- TJ[2:6]
- TJ[-6:5:2]

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i:]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do separador, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um separador `n` vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros     {0:05}  
idem: 3 casas após a vírgula   {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você deseja incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `PV="ERJXCXETA"`
Efetue todos os fatiamentos.
a) `PV[-7:8:2]`
b) `PV[3:6:2]`
c) `PV[1:8:2]`
d) `PV[:7:2]`
e) `PV[1:-1:3]`

2. Considere `DM="JUXTYWLI"`
Efetue todos os fatiamentos.
a) `DM[1:8:2]`
b) `DM[-7:6:3]`
c) `DM[1:-2]`
d) `DM[-6:7:2]`
e) `DM[3:7]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e



Strings em Python

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'  
'spam eggs'  
>>> 'doesn't'  
"doesn't"  
>>> "doesn't"  
"doesn't"  
>>> "Yes," he said.'  
'Yes," he said.'
```

Strings podem ser acessados como listas, mas eles são imutáveis, não podem ser alterados. Para alterar letras em strings, deve-se antes transformar o string em lista. Veja a ida e a volta

```
>>> L = list("Curitiba")  
>>> L[0]='X'  
>>> print(L)  
['X', 'u', 'r', 'i', 't', 'i', 'b', 'a']  
>>> s=''.join(L)  
>>> s  
'Xuritiba'  
>>> print(s)  
Xuritiba
```

Como se viu a função `list` transforma cada caractere da string em um elemento da lista retornada. O método `join` faz o inverso, transformando uma lista em um string.

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\  
diversas linhas assim como se faria em C.\n"
```

Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos. É exatamente o mesmo conceito estudado lá nas listas.

```
>>> palavra[4]  
'a'  
>>> palavra[0:2]  
'Aj'  
>>> palavra[2:4]  
'ud'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'  
=== Erro ===  
>>> palavra[:1] = 'Splat'  
=== Erro ===
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]  
'xjudaZ'  
>>> 'Splat' + palavra[5]  
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i:]` é igual a `s`.

Intervalos fora de limites: um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]  
'judaZ'  
>>> palavra[10:]  
''
```

```
>>> palavra[2:1]  
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Observe que `-0` é o mesmo que `0`, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]  
'A'
```

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'  
>>> len(s)  
29
```

Métodos de String

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize() Transforma o primeiro caractere de uma string em maiúsculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()  
'Curitiba'  
>>> "ágora".capitalize()  
ágora
```

.center() Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)  
'      paraná      '  
>>> "paraná".center(2)  
'paraná'
```

.count() Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

.find() Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna `-1`.

```
>>> "abracadabra".count("ab")  
2  
>>> "abracadabra".count("a")  
5
```

Avaliadores São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`, `.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join() Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']  
>>> ''.join(a)  
'aguafogoxabu'  
>>> '\n'.join(a)  
'agua\nfogo\nxabu'  
>>> print('\n'.join(a))  
agua  
fogo  
xabu
```

.partition() Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o `separador` e a parte posterior ao `separador`

```
>>> "curitibaparanabrasil".partition('ra')  
('curitibapa', 'ra', 'nabrasil')  
>>> "curitibaparanabrasil".partition('z')  
('curitibaparanabrasil', '', '')
```

.replace() Substitui `n` ocorrências da substring antiga pela nova. Se `n` não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra", "agora")  
'curitibapaagoranabagorasil'  
>>> "curitibaparanabrasil".replace("ra", "zz", 1)  
'curitibapazznabrasil'
```

.split() Divide a string transformando-a em uma lista usando um `separador` `n` vezes. Se o `separador` não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')  
['curitib', 'p', 'r', 'n', 'br', 'sil']  
>>> "curitiba parana brasil".split()  
['curitiba', 'parana', 'brasil']
```

Muitos outros Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

Formatação

Além da formatação padrão C (com `%d`, `%s`, `%f`, etc), Python introduziu a formatação representando os parâmetros por números em chaves. Veja o exemplo

```
>>> "{0} traga {1} aqui".format("Ze", 4)  
'Ze traga 4 aqui'
```

Graças a esta facilidade, pode-se repetir o mesmo valor na frase e reordenar os parâmetros. Eis algumas coisas que se podem fazer

```
estabelecer o tamanho           {0:10}  
alinhar à esquerda              {0:<10}  
centralizar no espaço           {0:^10}  
preencher o espaço com .       {0:.<10}  
número: tamanho 5 com zeros     {0:05}  
idem: 3 casas após a vírgula    {0:10.3}  
com + ou -                      {0:~10}{1:+3}  
em binário                      {0:b}  
em hexadecimal*{0:x} {1:X}
```

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja <http://www.unicode.org/>) e se integra bem aos objetos string pré-existent, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software. Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'  
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'  
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal `0x0020` (o espaço em branco) naquela posição.

🔗 Para você fazer

1. Considere `QT="OMAVCGYT"`
Efetue todos os fatiamentos.
 - a) `QT[1:-1:2]`
 - b) `QT[3:-1:2]`
 - c) `QT[1:7]`
 - d) `QT[3:6:3]`
 - e) `QT[7:2]`

2. Considere `LA="GBJKESPV"`
Efetue todos os fatiamentos.
 - a) `LA[-7:8:3]`
 - b) `LA[2:6]`
 - c) `LA[1:6:2]`
 - d) `LA[3:8:2]`
 - e) `LA[2:7]`

Responda aqui:

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e

