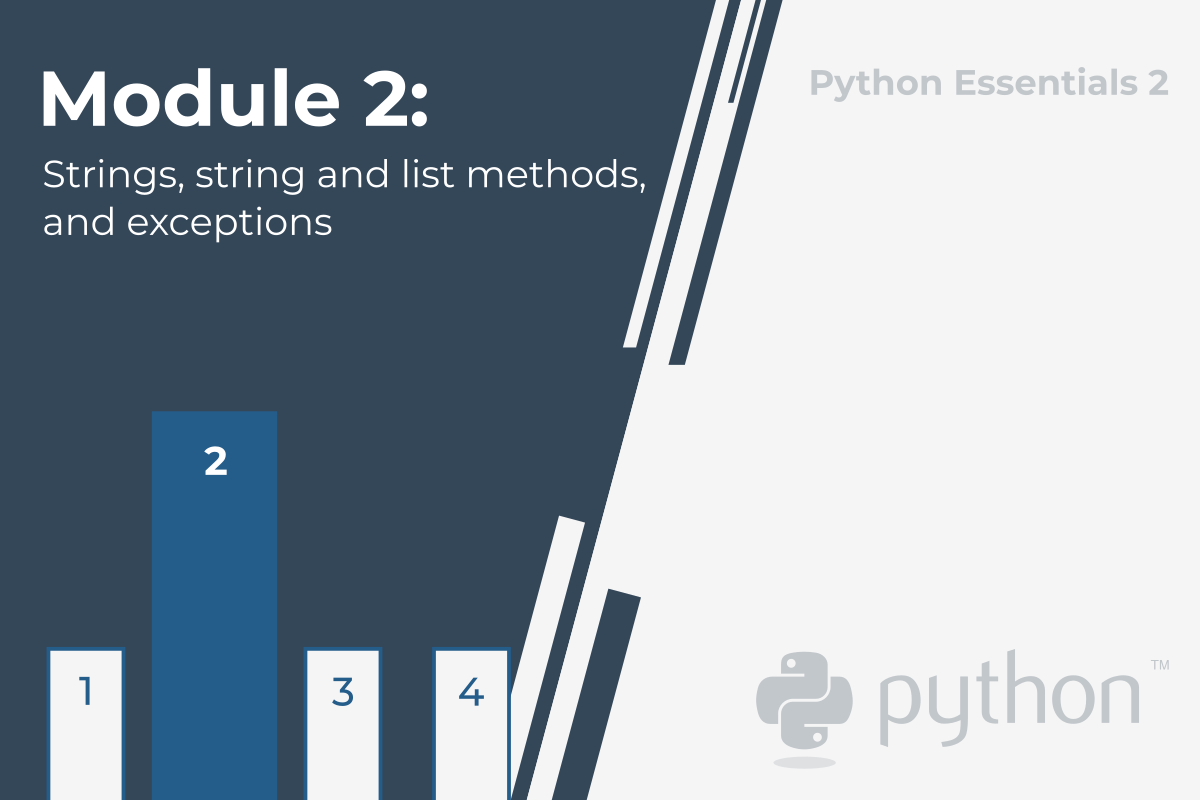
**Python Essentials 2:  
Módulo 2**

**Strings, Métodos de String e de Lista, Exceções**

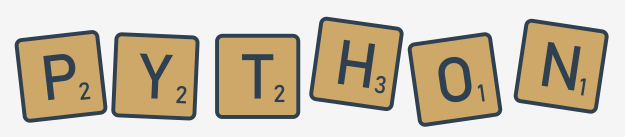
Neste módulo, aprenderá sobre:

* Carateres, strings e padrões de codificação;
* Strings vs. listas — semelhanças e diferenças;
* Métodos de Listas;
* Métodos de String;
* A forma do Python de lidar com erros de runtime;
* Controlar o fluxo de erros usando try e except;
* Hierarquia de exceções.



# Como os computadores entendem carateres individuais

Escreveu alguns programas interessantes desde que iniciou este curso, mas todos eles processaram apenas um tipo de dados - números. Como sabe (pode ver isto em todo o lado à sua volta) muitos dados informáticos não são números: nomes próprios, apelidos, moradas, títulos, poemas, artigos científicos, e-mails, sentenças judiciais, confissões de amor, e muito, muito mais.



Todos estes dados devem ser armazenados, introduzidos, enviados, pesquisados e transformados por computadores contemporâneos como qualquer outro dado, independentemente de serem carateres únicos ou enciclopédias multi-volume.

Como é isso possível?

Como pode fazer isto em Python? É o que vamos discutir agora. Comecemos pela forma como os computadores entendem os carateres individuais.

**Os computadores armazenam carateres como números**. Cada caratere utilizado por um computador corresponde a um número único, e vice-versa. Esta atribuição deve incluir mais carateres do que se poderia esperar. Muitos deles são invisíveis para os humanos, mas essenciais para os computadores.

Alguns destes carateres são chamados **whitespaces** (espaços em branco), enquanto outros são denominados **carateres de controlo**, porque o seu objetivo é controlar dispositivos de input/output.

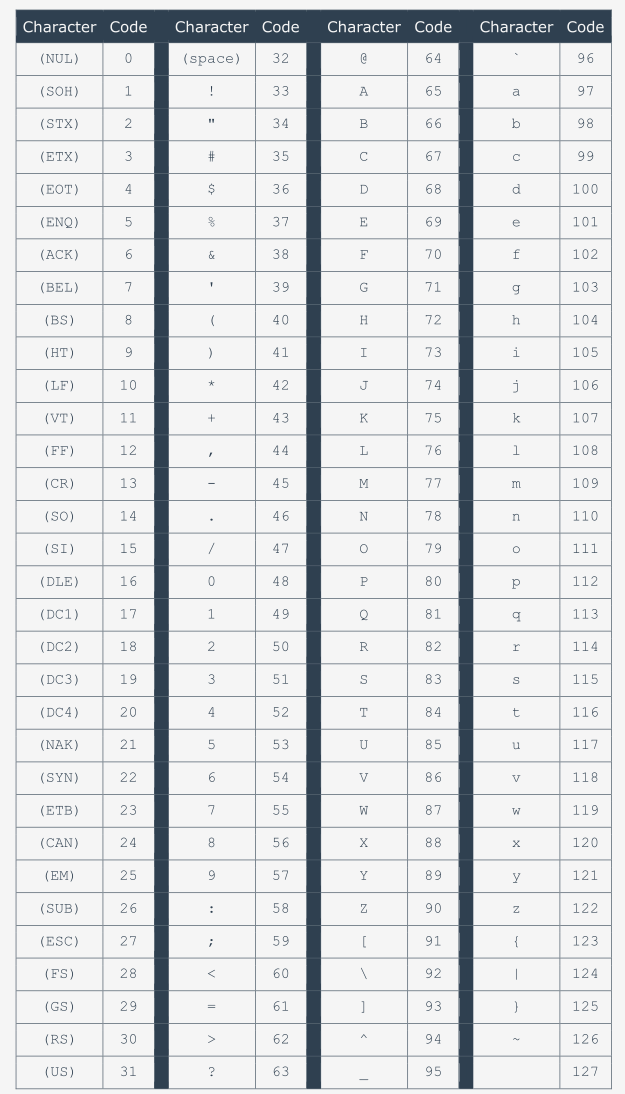
Um exemplo de um whitespace que é completamente invisível a olho nu é um código especial, ou um par de códigos (sistemas operativos diferentes podem tratar esta questão de forma diferente), que são utilizados para marcar as extremidades das linhas dentro dos ficheiros de texto.

As pessoas não vêem este sinal (ou estes sinais), mas são capazes de observar o efeito da sua aplicação onde as linhas são quebradas.

Podemos criar praticamente qualquer número de atribuições de números de carateres, mas a vida num mundo em que cada tipo de computador utiliza uma codificação de carateres diferente não seria muito conveniente. Este sistema levou à necessidade de introduzir uma norma universal e amplamente aceite implementada por (quase) todos os computadores e sistemas operativos em todo o mundo.

O chamado **ASCII** (abreviação de **American Standard Code for Information Interchange**) é o mais utilizado, e pode-se assumir que quase todos os dispositivos modernos (como computadores, impressoras, telemóveis, tablets, etc.) usam este código.

O código fornece espaço para **256 carateres diferentes**, mas estamos interessados apenas nos primeiros 128. Se quiser ver como o código é construído, veja a tabela em baixo. Clique na tabela para ampliá-la. Olha bem para ela - há alguns factos interessantes. Olhe para o código do caratere mais comum - o *espaço*. Isto é *32*.



Agora veja o código da letra minúscula *a*. Isto é *97*. E agora encontre a maiúscula *A*. O seu código é *65*. Agora descubra a diferença entre o código de *a* e de *A*. É igual a *32*. Esse é o código de um *espaço*. Interessante, não é?

Note-se também que as letras estão dispostas na mesma ordem que no alfabeto latino.

# I18N

É claro que o alfabeto latino não é suficiente para toda a humanidade. Os utilizadores desse alfabeto estão em minoria. Foi necessário inventar algo mais flexível e capacitivo do que o ASCII, algo capaz de tornar todo o software do mundo passível de **internacionalização**, porque línguas diferentes usam alfabetos completamente diferentes, e por vezes estes alfabetos não são tão simples como o latino.

A palavra *internationalization* (internacionalização) é normalmente abreviada para **I18N**.



Porquê? Olhe cuidadosamente - há um *I* na frente da palavra, a seguir há *18* letras diferentes, e um *N* no final.

Apesar da origem ligeiramente humorística, o termo é oficialmente utilizado em muitos documentos e normas.

O **software I18N** é uma norma nos tempos atuais. Cada programa tem de ser escrito de forma a poder ser utilizado em todo o mundo, entre diferentes culturas, línguas e alfabetos.

**Uma forma clássica de código ASCII utiliza oito bits para cada sinal**. Oito bits significam 256 carateres diferentes. Os primeiros 128 são utilizados para o alfabeto latino padrão (tanto carateres maiúsculos como minúsculos). É possível empurrar todos os outros carateres nacionais utilizados em todo o mundo para os 128 locais restantes?

Não. Não é.

## Code points e code pages

Precisamos agora de um novo termo: um **code point**.

Um code point é **um número que faz um caratere**. Por exemplo, *32* é um code point que faz um *espaço* na codificação ASCII. Podemos dizer que o código ASCII padrão consiste em 128 code points.

Como o ASCII padrão ocupa 128 dos 256 code points possíveis, só se pode fazer uso dos 128 restantes.

Não é o suficiente para todas as línguas possíveis, mas pode ser suficiente para uma língua, ou para um pequeno grupo de línguas semelhantes.

Pode-se **definir a metade superior dos code points de forma diferente para línguas diferentes**? Sim, pode-se. Tal conceito é chamado **code page**.

Um code page é uma **norma para utilizar os 128 code points superiores para armazenar carateres nacionais específicos**. Por exemplo, existem diferentes code pages para a Europa Ocidental e Europa Oriental, alfabetos cirílico e grego, árabe e hebraico, e assim por diante.

Isto significa que o mesmo code point pode fazer carateres diferentes quando usado em code pages diferentes.

Por exemplo, o code point *200* faz Č (uma letra utilizada por algumas línguas eslavas) quando utilizada pela página de código ISO/CEI 8859-2, e faz Ш (uma letra cirílica) quando utilizada pela página de código ISO/CEI 8859-5.

Em consequência, para determinar o significado de um code point específico, é necessário conhecer o code page alvo.

Por outras palavras, os code points derivados do conceito de code page são ambíguos.

# Unicode

As code pages ajudaram a indústria informática a resolver questões I18N durante algum tempo, mas depressa se revelou que não seriam uma solução permanente.

O conceito que resolveu o problema a longo prazo foi o **Unicode**.



**O Unicode atribui carateres únicos (inequívocos) (letras, hífens, ideogramas, etc.) a mais de um milhão de code points**. Os primeiros 128 code points de Unicode são idênticos aos ASCII, e os primeiros 256 code points de Unicode são idênticos à página de código ISO/IEC 8859-1 (uma página de código concebida para as línguas da Europa Ocidental).

## UCS-4

A norma Unicode não diz nada sobre como codificar e armazenar os carateres na memória e nos ficheiros. Apenas nomeia todos os carateres disponíveis e atribui-os a planos (um grupo de carateres de origem, aplicação, ou natureza semelhante).



Existe mais do que uma norma que descreve as técnicas utilizadas para implementar Unicode em computadores e sistemas de armazenamento de computadores reais. O mais geral deles é o **UCS-4**.

O nome vem de **Universal Character Set** (Conjunto de Carateres Universal).

**UCS-4 utiliza 32 bits (quatro bytes) para armazenar cada caratere**, e o código é apenas o número único dos code points Unicode. Um ficheiro contendo texto codificado UCS-4 pode começar com uma BOM (Byte Order Mark), uma combinação de bits não imprimível que anuncia a natureza do conteúdo do ficheiro. Algumas utilidades podem requerê-lo.

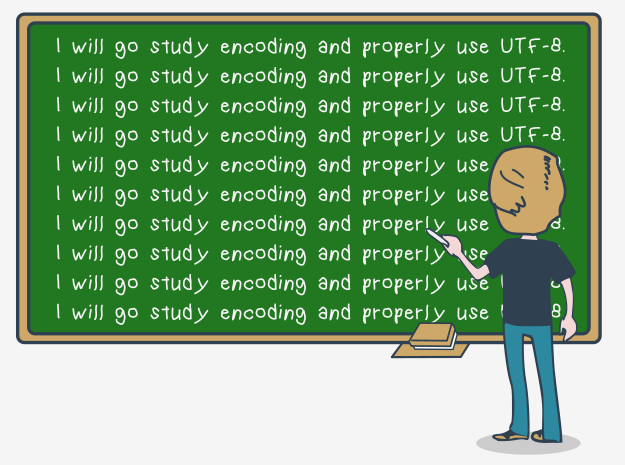
Como pode ver, o UCS-4 é uma norma bastante esbanjadora - aumenta o tamanho de um texto em quatro vezes em comparação com a norma ASCII. Felizmente, existem formas mais inteligentes de codificar textos Unicode.

## UTF-8

Um dos mais utilizados é o **UTF-8**.

O nome é derivado de **Unicode Transformation Format** (Formato de Transformação Unicode).

O conceito é muito inteligente. **O UTF-8 utiliza tantos bits para cada um dos code points quantos forem realmente necessários para os representar**.



Por exemplo:

* todos os carateres latinos (e todos os caracteres ASCII) ocupam oito bits;
* Carateres não-latinos ocupam 16 bits;
* Os ideógrafos da CJK (China-Japão-Coreia) ocupam 24 bits.

Devido às características do método utilizado pelo UTF-8 para armazenar os code points, não há necessidade de utilizar a lista técnica, mas algumas das ferramentas procuram-na quando leem o ficheiro, e muitos editores configuraram-na durante a gravação.

Python 3 suporta totalmente Unicode e UTF-8:

* pode utilizar carateres codificados Unicode/UTF-8 para nomear variáveis e outras entidades;
* pode utilizá-los durante todos os inputs e outputs.

Isto significa que o Python 3 é completamente I18Ned.

# Key takeaways

1. Os computadores armazenam carateres como números. Existe mais do que uma forma possível de codificar carateres, mas apenas alguns deles ganharam popularidade mundial e são normalmente utilizados em IT: são **ASCII** (utilizado principalmente para codificar o alfabeto latino e alguns dos seus derivados) e **UNICODE** (capaz de codificar virtualmente todos os alfabetos utilizados pelos seres humanos).

2. Um número correspondente a um determinado caratere é chamado um **codepoint**.

3. UNICODE utiliza diferentes formas de codificação quando se trata de armazenar os carateres utilizando ficheiros ou memória do computador: dois deles são **UCS-4** e **UTF-8** (este último é o mais comum, uma vez que desperdiça menos espaço de memória).

**Exercício 1**

O que é BOM?

**BOM** (Byte Order Mark) é uma combinação especial de bits que anunciam a codificação, utilizados pelo conteúdo de um ficheiro (por exemplo, UCS-4 ou UTF-B/8).

**Exercício 2**

O Python 3 é *I18N*do?

Sim, é completamente internacionalizado - podemos usar carateres UNICODE dentro do nosso código, lê-los a partir do input e enviá-los para o output.

**Strings - uma breve revisão**

Vamos fazer uma breve revisão sobre a natureza das strings do Python.

Em primeiro lugar, as strings do Python (ou simplesmente strings, pois não vamos discutir as strings de qualquer outra linguagem) são **sequências imutáveis**.

É muito importante notar isto, porque significa que se deve esperar algum comportamento familiar por parte delas.

Vamos analisar o código no editor para perceber do que estamos a falar:

* Dê uma vista de olhos no **Exemplo 1**. A função len() utilizada para strings devolve um número de carateres contidos pelos argumentos. O snippet faz o output 2.
* Qualquer string pode estar vazia. O seu comprimento é de 0 então - assim como no **Exemplo 2**.
* Não se esqueça que uma barra invertida (\) utilizada como caratere de escape não está incluída no comprimento total da string. O código no **Exemplo 3**, portanto, gera o output 3.

Execute os três códigos de exemplo e verifique.

# Example 1

word = 'by'

print(len(word))

# Example 2

empty = ''

print(len(empty))

# Example 3

i\_am = 'I\'m'

print(len(i\_am))

# Strings multiline

Agora é um momento muito bom para lhe mostrar outra forma de especificar strings dentro do source code de Python. Note que a sintaxe que já conhece não o deixará usar uma string que ocupa mais do que uma linha de texto.

Por esta razão, o código aqui está errado:

multiline = 'Line #1

Line #2'

print(len(multiline))

Felizmente, para este tipo de strings, o Python oferece uma sintaxe separada, conveniente e simples.

Veja o código no editor. É este o seu aspeto.

multiline = '''Line #1

Line #2'''

print(len(multiline))

Como pode ver, a string começa com **três apóstrofes**, não uma. A mesma apóstrofe tripla é utilizada para a terminar.

O número de linhas de texto colocadas dentro de tal string é arbitrário.

O snippet faz o output 15.

Conte os carateres com cuidado. Este resultado está correto ou não? À primeira vista parece estar, mas quando se conta os carateres não está.

Line #1 contém sete carateres. Duas dessas linhas compreendem 14 carateres. Perdemos um caratere? Onde? Como?

Não, não perdemos.

**O caratere que falta é simplesmente invisível - é um espaço em branco**. Ele está localizado entre as duas linhas de texto.

É denotado como: \n.

Lembra-se? É um caratere especial (de controlo) utilizado para **forçar uma line feed** (alimentação de linha) (daí o seu nome: LF). Não se consegue vê-lo, mas conta.

As strings multiline também podem ser delimitadas por **aspas triplas**, tal como aqui:

multiline = """Line #1

Line #2"""

print(len(multiline))

Escolha o método que lhe seja mais confortável. Ambos funcionam da mesma forma.

**Operações em strings**

Como outros tipos de dados, as strings têm o seu próprio conjunto de operações permitidas, embora sejam bastante limitadas em comparação com os números.

Em geral, as strings podem ser:

* **concatenadas** (juntas)
* **replicadas**.

A primeira operação é realizada pelo operador + (note: não é uma adição) enquanto a segunda pelo operador \* (note novamente: não é uma multiplicação).

A capacidade de utilizar o mesmo operador contra tipos de dados completamente diferentes (como números vs. strings) chama-se **overloading** (visto tal operador estar sobrecarregado com tarefas diferentes).

Analise o exemplo:

* O operador + utilizado contra duas ou mais strings produz uma nova string contendo todos os carateres dos seus argumentos (nota: a ordem importa - este +overloaded, em contraste com a sua versão numérica, **não é comutativa**)
* o operador \* precisa de uma string e um número como argumentos; neste caso, a ordem não importa - pode colocar o número antes da string, ou vice-versa, o resultado será o mesmo - uma nova string criada pela enésima replicação da string do argumento.

O snippet produz o seguinte output:

ab

ba

aaaaa

bbbb

**output**

Nota: as variantes de atalho dos operadores acima referidos são também aplicáveis para strings (+= e \*=).

# Operações em strings: ord()

Se quiser **saber o valor do code point ASCII/UNICODE de um caratere específico**, pode usar uma função chamada ord() (como em *ordinal*).

A função precisa de **uma string de um único caratere como seu argumento** - violar este requisito causa uma exceção TypeError , e devolve um número que representa o code point do argumento.

Veja o código no editor, e execute-o.

# Demonstrating the ord() function.

char\_1 = 'a'

char\_2 = ' ' # space

print(ord(char\_1))

print(ord(char\_2))

O snippet faz o output:

97

32

**output**

Agora atribua valores diferentes para char\_1 e char\_2, por exemplo, α (alfa grego), e ę (uma letra no alfabeto polonês); em seguida execute o código e veja o resultado que ele produz. Realize as suas próprias experiências.

**Operações em strings: chr()**

Se conhece o code point (número) e pretende obter o caratere correspondente, pode utilizar uma função chamada chr().

A função **toma um code point e devolve o seu caratere**.

Invocá-lo com um argumento inválido (por exemplo, um code point negativo ou inválido) causa ValueError ou TypeError exceções.

Execute o código no editor. O output do snippet de exemplo:

a

α

**output**

Nota:

* chr(ord(x)) == x
* ord(chr(x)) == x

Mais uma vez, faça as suas próprias experiências.

# Strings como sequências: indexação

Dissemos-lhe antes que as **strings de Python são sequências**. É tempo de lhe mostrar o que isso realmente significa.

As strings não são listas, mas **pode tratá-las como listas em muitos casos particulares**.

Por exemplo, se quiser aceder a qualquer um dos carateres de uma string, pode fazê-lo utilizando a **indexação**, tal como no exemplo abaixo. Execute o programa:

# Indexing strings.

the\_string = 'silly walks'

for ix in range(len(the\_string)):

print(the\_string[ix], end=' ')

print()

Tenha cuidado - não tente passar os limites de uma string - isso causará uma exceção.

O output do exemplo é:

s i l l y w a l k s

**output**

A propósito, os índices negativos também se comportam como esperado. Verifique isto você mesmo.

# Strings como sequências: iteração

**Iterar através das strings** também funciona. Veja o exemplo abaixo:

# Iterating through a string.

the\_string = 'silly walks'

for character in the\_string:

print(character, end=' ')

print()

O output é o mesmo que anteriormente. Verifique.

# Slices

Além disso, tudo o que sabe sobre **slices** ainda é utilizável.

Reunimos alguns exemplos que mostram como as slices funcionam no mundo das strings. Olhe para o código no editor, analise-o e execute-o.

Não verá nada de novo no exemplo, mas queremos que tenha a certeza de que pode explicar todas as linhas do código.

O output do código é:

bd

efg

abd

e

e

adf

beg

**output**

Agora faça as suas próprias experiências.

# Os loops Operadores in e not in .

**O operador in .**

O operador in não deve surpreendê-lo quando aplicado a strings - simplesmente **verifica se o seu argumento esquerdo (uma string) pode ser encontrado em qualquer lugar dentro do argumento direito (outra string)**.

O resultado da verificação é simplesmente True ou False.

Veja o programa de exemplo abaixo. É assim que o operador in funciona:

alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" in alphabet)

print("F" in alphabet)

print("1" in alphabet)

print("ghi" in alphabet)

print("Xyz" in alphabet)

O output do exemplo é:

True

False

False

True

False

**output**

**O método not in .**

Como provavelmente suspeita, o operador not in também é aplicável aqui.

É assim que funciona:

alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" not in alphabet)

print("F" not in alphabet)

print("1" not in alphabet)

print("ghi" not in alphabet)

print("Xyz" not in alphabet)

O output do exemplo é:

False

True

True

False

True

**output**

# As strings de Python são imutáveis

Também lhe dissemos que **as strings de Python são imutáveis**. Esta é uma característica muito importante. O que significa isto?

Isto significa principalmente que a semelhança de strings e listas é limitada. Nem tudo o que se pode fazer com uma lista pode ser feito com uma string.

A primeira diferença importante **não lhe permite utilizar a instrução**del**para remover qualquer coisa de uma string**.

O exemplo aqui não vai funcionar:

alphabet = "abcdefghijklmnopqrstuvwxyz"

del alphabet[0]

A única coisa que pode fazer com del e uma string é **remover a string como um todo**. Tente fazê-lo.

As strings de Python **não têm o método**append()- não se pode expandi-las de forma alguma.

O exemplo abaixo está errado:

alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.append("A")

com a ausência do método append() , **o método**insert()**é ilegal**, também:

alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.insert(0, "A")

Não pense que a imutabilidade de uma string limita a sua capacidade de operar com strings.

A única consequência é que tem de se lembrar disso, e implementar o seu código de uma forma ligeiramente diferente - veja o exemplo do código no editor.

Esta forma de código é totalmente aceitável, funcionará sem contornar as regras do Python, e trará o alfabeto latino completo para o seu ecrã:

abcdefghijklmnopqrstuvwxyz

**output**

Poderá perguntar-se se **a criação de uma nova cópia de uma string, cada vez que modifica o seu conteúdo, piora a eficácia do código**.

Sim, piora. Um pouco. No entanto, não é de todo um problema.

# Operações em strings: min()

Agora que compreende que as strings são sequências, podemos mostrar-lhe algumas capacidades de sequência menos óbvias. Vamos apresentá-las utilizando strings, mas não se esqueça de que as listas também podem adotar os mesmos truques.

# Demonstrating min() - Example 1:

print(min("aAbByYzZ"))

# Demonstrating min() - Examples 2 &amp; 3:

t = 'The Knights Who Say "Ni!"'

print('[' + min(t) + ']')

t = [0, 1, 2]

print(min(t))

Vamos começar com uma função chamada min().

A função **encontra o elemento mínimo da sequência passada como um argumento**. Há uma condição - a sequência (string, lista, não importa) **não pode estar vazia**, ou então terá uma exceção ValueError .

O programa do **Exemplo 1** tem como output:

A

**output**

Nota: É uma maiúscula *A*. Porquê? Lembre-se da tabela ASCII - que letras ocupam os primeiros locais - superior ou inferior?

Preparámos mais dois exemplos para analisar: **Exemplos 2 & 3**.

Como pode ver, eles apresentam mais do que apenas strings. O output esperado parece-se com o seguinte:

[ ]

0

**output**

Nota: usamos os parêntesis retos para evitar que o espaço seja negligenciado no ecrã.

# Operações em strings: max()

Da mesma forma, uma função chamada max() **encontra o elemento máximo da sequência**.

# Demonstrating max() - Example 1:

print(max("aAbByYzZ"))

# Demonstrating max() - Examples 2 &amp; 3:

t = 'The Knights Who Say "Ni!"'

print('[' + max(t) + ']')

t = [0, 1, 2]

print(max(t))

Veja o **Exemplo 1** no editor. O output do programa de exemplo:

z

**output**

Nota: É um *z* minúsculo.

Agora vamos ver a função max() aplicada aos mesmos dados que anteriormente. Veja os **Exemplos 2 & 3** no editor.

O output esperado é:

[y]

2

**output**

Realize as suas próprias experiências.

# Operações em strings: o método index() .

A classe index() (é um método, não uma função) **pesquisa a sequência desde o início, a fim de encontrar o primeiro elemento do valor especificado no seu argumento**.

Nota: o elemento pesquisado deve ocorrer na sequência - **a sua ausência causará uma exceção ValueError**.

O método devolve o **index da primeira ocorrência do argumento** (o que significa que o menor resultado possível é 0, enquanto o maior é o comprimento do argumento decrescido por 1).

# Demonstrating the index() method:

print("aAbByYzZaA".index("b"))

print("aAbByYzZaA".index("Z"))

print("aAbByYzZaA".index("A"))

Portanto, o exemplo do editor dará o output:

2

7

1

**output**

# Operações em strings: a função list() .

A função list() **toma o seu argumento (uma string) e cria uma nova lista contendo todos os carateres da string, um por elemento de lista**.

Nota: não é estritamente uma função string - list() é capaz de criar uma nova lista de muitas outras entidades (por exemplo, de tuples e dicionários).

Dê uma vista de olhos no exemplo de código no editor.

# Demonstrating the list() function:

print(list("abcabc"))

O output do exemplo:

['a', 'b', 'c', 'a', 'b', 'c']

**output**

# Operações em strings: o método count() .

A classe count() **conta todas as ocorrências do elemento dentro da sequência**. A ausência de tais elementos não causa problemas.

Veja o segundo exemplo no editor.

# Demonstrating the count() method:

print("abcabc".count("b"))

print('abcabc'.count("d"))

Consegue adivinhar o seu output?

É:

2

0

**output**

Além disso, as strings em Python têm um número significativo de métodos destinados exclusivamente ao processamento de carateres. Não espere que funcionem com quaisquer outras coleções. A lista completa é apresentada aqui: <https://docs.python.org/3.4/library/stdtypes.html#string-methods>.

Vamos mostrar-lhe as que consideramos mais úteis.

**Key takeaways**

1. As strings de Python são **sequências imutáveis** e podem ser indexadas, sliced (cortadas) e iteradas como qualquer outra sequência, além de serem sujeitas aos operadores in e not in . Existem dois tipos de strings em Python:

* strings **one-line**, que não podem cruzar limites de linha - denotamo-las usando apóstrofes ('string') ou aspas ("string")
* strings **multi-line**, que ocupam mais de uma linha de source code, delimitadas por trígrafos:
* '''
* string
* '''

ou

"""

string

"""

2. O comprimento de uma string é determinado pela função len() . O caratere de escape (\) não é contado. Por exemplo:

print(len("\n\n"))

tem como output 2.

3. As strings podem ser **concatenadas** usando o operador + e **replicadas** usando o operador \* . Por exemplo:

asterisk = '\*'

plus = "+"

decoration = (asterisk + plus) \* 4 + asterisk

print(decoration)

tem como output \*+\*+\*+\*+\*.

4. O par de funções chr() e ord() pode ser utilizado para criar um caratere usando o seu codepoint, e para determinar um codepoint correspondente a um caratere. Ambas as expressões a seguir são sempre verdadeiras:

chr(ord(character)) == character

ord(chr(codepoint)) == codepoint

5. Algumas outras funções que podem ser aplicadas a strings são:

* list() — criar uma lista composta por todos os carateres da string;
* max() — encontrar o caratere com o codepoint máximo;
* min() — encontrar o caratere com o codepoint mínimo;

6. O método chamado index() encontra o index de uma determinada substring dentro da string.

**Exercício 1**

Qual é o comprimento da seguinte string supondo que não haja espaços em branco entre as aspas?

"""

"""

Verifique

**Exercício 2**

Qual é o output esperado do seguinte código?

s = 'yesteryears'

the\_list = list(s)

print(the\_list[3:6])

Verifique

**Exercício 3**

Qual é o output esperado do seguinte código?

for ch in "abc":

print(chr(ord(ch) + 1), end='')

**Os loops capitalize() .**

Vamos passar por alguns métodos padrão de string Python. Vamos atravessá-los por ordem alfabética - para ser honesto, qualquer ordem tem tantas desvantagens como vantagens, pelo que a escolha pode também ser aleatória.

O método capitalize() faz exatamente o que diz - **cria uma nova string cheia de carateres retirados da source string**, mas tenta modificá-los da seguinte forma:

* **se o primeiro caratere dentro da string for uma letra** (nota: o primeiro caratere é um elemento com um index igual a 0, não apenas o primeiro caratere visível), **será convertido para maiúsculas**;
* **todas as letras restantes da string serão convertidas em minúsculas**.

Não se esqueça:

* a string original (da qual o método é invocado) não é alterada de forma alguma (a imutabilidade de uma string deve ser obedecida sem reservas)
* a string modificada (capitalizada neste caso) é devolvida como resultado - se não a utilizar de qualquer forma (atribuí-la a uma variável, ou passá-la a uma função/método) ela desaparecerá sem deixar rasto.

Nota: os métodos não têm de ser invocados apenas de dentro das variáveis. Podem ser invocados diretamente de dentro de literais de string. Vamos utilizar regularmente essa convenção - simplificará os exemplos, uma vez que os aspetos mais importantes não desaparecerão entre as tarefas desnecessárias.

Veja o exemplo no editor. Execute-o.

# Demonstrating the capitalize() method:

print('aBcD'.capitalize())

Isto é o que ele imprime:

Abcd

**output**

Experimente alguns exemplos mais avançados e teste o seu output:

print("Alpha".capitalize())

print('ALPHA'.capitalize())

print(' Alpha'.capitalize())

print('123'.capitalize())

print("αβγδ".capitalize())

# Os loops center() .

A variante de um parâmetro do método center() faz uma cópia da string original, tentando centralizá-la dentro de um campo de uma largura especificada.

A centralização é realmente feita **adicionando alguns espaços antes e depois da string**.

Não espere que este método demonstre quaisquer competências sofisticadas. É bastante simples.

O exemplo no editor usa parêntesis retos para mostrar claramente onde começa e termina a string centrada.

# Demonstrating the center() method:

print('[' + 'alpha'.center(10) + ']')

O seu output é o seguinte:

[ alpha ]

**output**

Se o comprimento do campo alvo for demasiado pequeno para caber na string, a string original é devolvida.

Pode ver o método center() em mais exemplos aqui:

print('[' + 'Beta'.center(2) + ']')

print('[' + 'Beta'.center(4) + ']')

print('[' + 'Beta'.center(6) + ']')

Execute os snippets acima e verifique qual o output que eles produzem.

**A variante de dois parâmetros de**center()**faz uso do caratere do segundo argumento, em vez de um espaço**. Analise o exemplo abaixo:

print('[' + 'gamma'.center(20, '\*') + ']')

É por isso que o output agora se assemelha a este:

[\*\*\*\*\*\*\*gamma\*\*\*\*\*\*\*\*]

**output**

Realize mais experiências.

# Os loops endswith() .

A classe endswith() **verifica se a string dada termina com o argumento especificado e devolve**True**ou**False, dependendo do resultado da verificação.

Nota: a substring deve aderir ao último caratere da string - não pode estar localizada apenas algures perto do final da string.

Veja o nosso exemplo no editor, analise-o, e execute-o. O seu output é:

yes

**output**

Deverá agora ser capaz de prever o output do snippet abaixo:

t = "zeta"

print(t.endswith("a"))

print(t.endswith("A"))

print(t.endswith("et"))

print(t.endswith("eta"))

Execute o código para verificar as suas previsões.

**Os loops find() .**

A classe find() é semelhante a index(), que já conhece - **procura uma substring e devolve o index de primeira ocorrência desta substring**, mas:

* é mais seguro - **não gera um erro para um argumento que contém uma substring inexistente** (devolve -1 então)
* **funciona apenas com strings** - não tente aplicá-lo a qualquer outra sequência.

Veja o código no editor. É assim que pode utilizá-lo.

O exemplo imprime:

1

-1

**output**

Nota: não use find() se quiser apenas verificar se um único caratere ocorre dentro de uma string - o operador in será significativamente mais rápido.

Aqui está outro exemplo:

t = 'theta'

print(t.find('eta'))

print(t.find('et'))

print(t.find('the'))

print(t.find('ha'))

Consegue prever o output? Execute-o e verifique as suas previsões.

Se quiser realizar a procura, não desde o início da string, mas **a partir de qualquer posição**, pode usar uma **variante de dois parâmetros** do find() método. Veja o exemplo:

print('kappa'.find('a', 2))

O segundo argumento **especifica o index em que a pesquisa será iniciada** (não tem de caber dentro da string).

Entre as duas letras *a*, apenas a segunda será encontrada. Execute o snippet e verifique.

Pode utilizar o método find() para procurar todas as ocorrências da substring, como aqui:

the\_text = """A variation of the ordinary lorem ipsum

text has been used in typesetting since the 1960s

or earlier, when it was popularized by advertisements

for Letraset transfer sheets. It was introduced to

the Information Age in the mid-1980s by the Aldus Corporation,

which employed it in graphics and word-processing templates

for its desktop publishing program PageMaker (from Wikipedia)"""

fnd = the\_text.find('the')

while fnd != -1:

print(fnd)

fnd = the\_text.find('the', fnd + 1)

O código imprime os índices de todas as ocorrências do artigo *the*, e o seu output é semelhante a este:

15

80

198

221

238

**output**

Há também uma **mutação de três parâmetros do método**find()- o terceiro argumento **aponta para o primeiro index que não será tomado em consideração durante a pesquisa** (na realidade é o limite superior da pesquisa).

Veja o nosso exemplo abaixo:

print('kappa'.find('a', 1, 4))

print('kappa'.find('a', 2, 4))

O segundo argumento especifica o index em que a pesquisa será iniciada (não tem de caber dentro da string).

Portanto, o exemplo modificado tem como output:

1

-1

**output**

(*a* não pode ser encontrado dentro dos limites de pesquisa indicados no segundo print().

# Os loops isalnum() .

(**IS** **AL**pha **NUM**eric)

O método sem parâmetros chamado isalnum() **verifica se a string contém apenas dígitos ou carateres alfabéticos (letras) e devolve**True**ou**False de acordo com o resultado.

Veja o exemplo no editor e execute-o.

Nota: qualquer elemento de string que não seja um dígito ou uma letra faz com que o método devolva False. Uma string vazia também o faz.

# Demonstrating the isalnum() method:

print('lambda30'.isalnum())

print('lambda'.isalnum())

print('30'.isalnum())

print('@'.isalnum())

print('lambda\_30'.isalnum())

print(''.isalnum())

O output do exemplo é:

True

True

True

False

False

False

**output**

Três exemplos mais intrigantes estão aqui:

t = 'Six lambdas'

print(t.isalnum())

t = 'ΑβΓδ'

print(t.isalnum())

t = '20E1'

print(t.isalnum())

Execute-os e verifique o seu output.

Dica: a causa do primeiro resultado é um espaço - não é nem um dígito nem uma letra.

# Os loops isalpha() .

A classe isalpha() é mais especializado - está interessado **apenas em letras**.

# Example 1: Demonstrating the isapha() method:

print("Moooo".isalpha())

print('Mu40'.isalpha())

# Example 2: Demonstrating the isdigit() method:

print('2018'.isdigit())

print("Year2019".isdigit())

Veja o Exemplo 1 - o seu output é:

True

False

**output**

# O método isdigit() .

DIGIT = Números

Por sua vez, o método isdigit() olha **apenas para os dígitos** - qualquer outra coisa produz False como o resultado.

Veja o Exemplo 2 - o seu output é:

True

False

**output**

Realize mais experiências.

# Os loops islower() .

A classe islower() é uma variante picuinhas do isalpha() - aceita **apenas letras minúsculas**.

# Example 1: Demonstrating the islower() method:

print("Moooo".islower())

print('moooo'.islower())

# Example 2: Demonstrating the isspace() method:

print(' \n '.isspace())

print(" ".isspace())

print("mooo mooo mooo".isspace())

# Example 3: Demonstrating the isupper() method:

print("Moooo".isupper())

print('moooo'.isupper())

print('MOOOO'.isupper())

Veja o Exemplo 1 no editor - o seu output é:

False

True

**output**

# O método isspace() .

A classe isspace() **identifica apenas os espaços em branco** - ignora qualquer outro caratere (o resultado é False então).

Veja o Exemplo 2 no editor - o seu output é:

True

True

False

**output**

# O método isupper() .

A classe isupper() é a versão maiúscula do islower() - concentra-se **apenas em letras maiúsculas**.

Novamente, veja o Exemplo 3 no editor - o seu output é:

False

False

True

# Os loops lower() .

A classe lower() **faz uma cópia de uma source string, substitui todas as letras maiúsculas por minúsculas**, e devolve a string como resultado. Novamente, a source string permanece intocada.

Se a string não contiver carateres maiúsculos, o método devolve a string original.

# Demonstrating the lower() method:

print("SiGmA=60".lower())

Nota: O método lower() não toma nenhum parâmetro.

O exemplo no editor tem como output:

sigma=60

**output**

Como de costume, realize as suas próprias experiências.

# Os loops lstrip () .

**L**eft **STRIP**

O parâmetro sem lstrip() método **devolve uma cadeia recém-criada formada a partir da original, removendo todos os principais espaços em branco**.

Analise o código de exemplo no editor.

# Demonstrating the lstrip() method:

print("[" + " tau ".lstrip() + "]")

Os parêntesis não fazem parte do resultado - apenas mostram os limites do resultado.

O output do exemplo:

[tau ]

O **parâmetro único** lstrip() faz o mesmo que a sua versão sem parâmetros, mas **remove todos os caracteres alistados no seu argumento** (uma cadeia), e não apenas espaços em branco:

print("www.cisco.com".lstrip("w."))

Não são necessários parêntesis aqui, pois o output é o seguinte:

cisco.com

**output**

Consegue adivinhar a saída do trecho abaixo? Pense bem. Execute o código e verifique as suas previsões.

print("pythoninstitute.org".lstrip(".org"))

Surpreendido? Carateres **principais** , espaços em branco principais. Mais uma vez, experimente com os seus próprios exemplos.

# Os loops replace() .

O método de **dois parâmetros** replace() **devolve uma cópia da string original na qual todas as ocorrências do primeiro argumento foram substituídas pelo segundo argumento**.

Veja o código de exemplo no editor. Execute-o.

# Demonstrating the replace() method:

print("www.netacad.com".replace("netacad.com", "pythoninstitute.org"))

print("This is it!".replace("is", "are"))

print("Apple juice".replace("juice", ""))

O output do exemplo:

www.pythoninstitute.org

Thare are it!

Apple

**output**

Se o segundo argumento for uma string vazia, **a substituição está na realidade a remover** a string do primeiro argumento. Que tipo de magia acontece se o primeiro argumento for uma string vazia?

A variante de **três parâmetros** replace() utiliza o terceiro argumento (um número) para **limitar o número de substituições**.

Veja o código de exemplo modificado abaixo:

print("This is it!".replace("is", "are", 1))

print("This is it!".replace("is", "are", 2))

Consegue adivinhar o seu output? Execute o código e verifique as suas suposições.

# Os loops rfind() .

Os métodos de um, dois e três parâmetros chamados rfind() fazem quase as mesmas coisas que os seus homólogos (os desprovidos do prefixo *r*), mas **começam as suas buscas a partir do fim da string**, não do início (daí o prefixo *r*, para *right*).

# Demonstrating the rfind() method:

print("tau tau tau".rfind("ta"))

print("tau tau tau".rfind("ta", 9))

print("tau tau tau".rfind("ta", 3, 9))

Veja o exemplo do código no editor e tente prever o seu output. Execute o código para verificar se estava certo.

# Os loops rstrip() .

**R**ight **STRIP**

Duas variantes do método rstrip() fazem quase o mesmo que lstrip, mas **afetam o lado oposto da string**.

# Demonstrating the rstrip() method:

print("[" + " upsilon ".rstrip() + "]")

print("cisco.com".rstrip(".com"))

Veja o exemplo de código no editor. Consegue adivinhar o seu output? Execute o código para verificar as suas suposições.

Como habitualmente, encorajamo-lo a experimentar os seus próprios exemplos.

# Os loops split() .

A classe split() faz o que diz - **divide a string e constrói uma lista de todas as substrings detetadas**.

O método **assume que as substrings são delimitadas por espaços em branco** - os espaços não participam na operação, e não são copiados para a lista resultante.

Se a string estiver vazia, a lista resultante também estará vazia.

# Demonstrating the split() method:

print("phi chi\npsi".split())

Veja o código no editor. O exemplo produz o seguinte output:

['phi', 'chi', 'psi']

**output**

Nota: a operação inversa pode ser realizada pelo método join() .

# Os loops startswith() .

A classe startswith() é um reflexo espelhado de endswith() - **verifica se uma determinada string começa com a substring especificada**.

# Demonstrating the startswith() method:

print("omega".startswith("meg"))

print("omega".startswith("om"))

print()

# Demonstrating the strip() method:

print("[" + " aleph ".strip() + "]")

Veja o exemplo no editor. Este é o resultado:

False

True

**output**

# O método strip() .

A classe strip() combina os efeitos causados por rstrip() e lstrip() - **faz uma nova string com falta de todos os espaços em branco à esquerda e à direita**.

Veja o segundo exemplo no editor. Este é o resultado que ele devolve:

[aleph]

**output**

Agora, realize as suas próprias experiências com os dois métodos.

# Os loops swapcase() .

A classe swapcase() **faz uma nova string, trocando a maiúscula/minúscula de todas as letras dentro da source string**: carateres minúsculos tornam-se maiúsculas e vice-versa.

Todos os outros carateres permanecem intocados.

# Demonstrating the swapcase() method:

print("I know that I know nothing.".swapcase())

print()

# Demonstrating the title() method:

print("I know that I know nothing. Part 1.".title())

print()

# Demonstrating the upper() method:

print("I know that I know nothing. Part 2.".upper())

Veja o primeiro exemplo no editor. Consegue adivinhar o output? Não terá bom aspeto, mas tem de o ver:

i KNOW THAT i KNOW NOTHING.

**output**

# O método title() .

A classe title() desempenha uma função algo semelhante - **muda a primeira letra de cada palavra para maiúscula, transformando todas as outras em minúsculas**.

Veja o segundo exemplo no editor. Consegue adivinhar o seu output? Este é o resultado:

I Know That I Know Nothing. Part 1.

**output**

# O método upper() .

Por último, mas não menos importante, o método upper() **faz uma cópia da source string, substitui todas as letras minúsculas pelas suas contrapartes maiúsculas**, e devolve a string como resultado.

Veja o terceiro exemplo no editor. O seu output é:

I KNOW THAT I KNOW NOTHING. PART 2.

**output**

Viva! Chegámos ao fim desta secção. Está surpreendido com algum dos métodos de string que discutimos até agora? Tire uns minutos para os rever, e passemos à próxima parte do curso onde lhe mostraremos as grandes coisas que podemos fazer com strings.

**Key takeaways**

1. Alguns dos métodos oferecidos por strings são:

* capitalize() — alterar todas as letras da string para maiúsculas;
* center() — centrar a string dentro do campo de um comprimento conhecido;
* count() — contar as ocorrências de um determinado caratere;
* join() — juntar todos os itens de uma tuple/lista numa string;
* lower() — converter todas as letras da string em letras minúsculas;
* lstrip() — remover os carateres brancos desde o início da string;
* replace() — substituir uma determinada substring por outra;
* rfind() — encontrar uma substring a partir do final da string;
* rstrip() — remover os espaços em branco à direita do final da string;
* split() — dividir a string numa substring usando um determinado delimitador;
* strip() — remover os espaços em branco à esquerda e à direita;
* swapcase() — trocar letras maiúsculas e minúsculas (minúsculas para maiúsculas e vice-versa)
* title() — tornar a primeira letra de cada palavra uma maiúscula;
* upper() — converter todas as letras da string em letras maiúsculas.

2. O conteúdo das strings pode ser determinado utilizando os seguintes métodos (todos eles devolvem valores Booleanos):

* endswith() — a string termina com uma determinada substring?
* isalnum() — a string consiste apenas em letras e dígitos?
* isalpha() — a string consiste apenas em letras?
* islower() — a string consiste apenas em letras minúsculas?
* isspace() — a string consiste apenas em espaços em branco?
* isupper() — a string consiste apenas em letras maiúsculas?
* startswith() — a string começa com uma determinada substring?

**Exercício 1**

Qual é o output esperado do seguinte código?

for ch in "abc123XYX":

if ch.isupper():

print(ch.lower(), end='')

elif ch.islower():

print(ch.upper(), end='')

else:

print(ch, end='')

Verifique

**Exercício 2**

Qual é o output esperado do seguinte código?

s1 = 'Where are the snows of yesteryear?'

s2 = s1.split()

print(s2[-2])

Verifique

**Exercício 3**

Qual é o output esperado do seguinte código?

the\_list = ['Where', 'are', 'the', 'snows?']

s = '\*'.join(the\_list)

print(s)

Verifique

**Exercício 4**

Qual é o output esperado do seguinte código?

s = 'It is either easy or impossible'

s = s.replace('easy', 'hard').replace('im', '')

print(s)

## LAB Seu próprio split

## Objetivos

* melhorar as habilidades do aluno a operar com strings;
* utilizar métodos de string Python incorporados.

## Cenário

Já sabe como split() funciona. Agora queremos que o prove.

A sua tarefa é **escrever a sua própria função, que se comporta quase exatamente como o método original**split(), ou seja:

* deve aceitar exatamente um argumento - uma string;
* deve devolver uma lista de palavras criadas a partir da string, dividida nos locais onde a string contém espaços em branco;
* se a string estiver vazia, a função deve devolver uma lista vazia;
* o seu nome deve ser mysplit()

Utilize o template no editor. Teste o seu código com cuidado.

## Output esperado

['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question']

['To', 'be', 'or', 'not', 'to', 'be,that', 'is', 'the', 'question']

[]

['abc']

[]