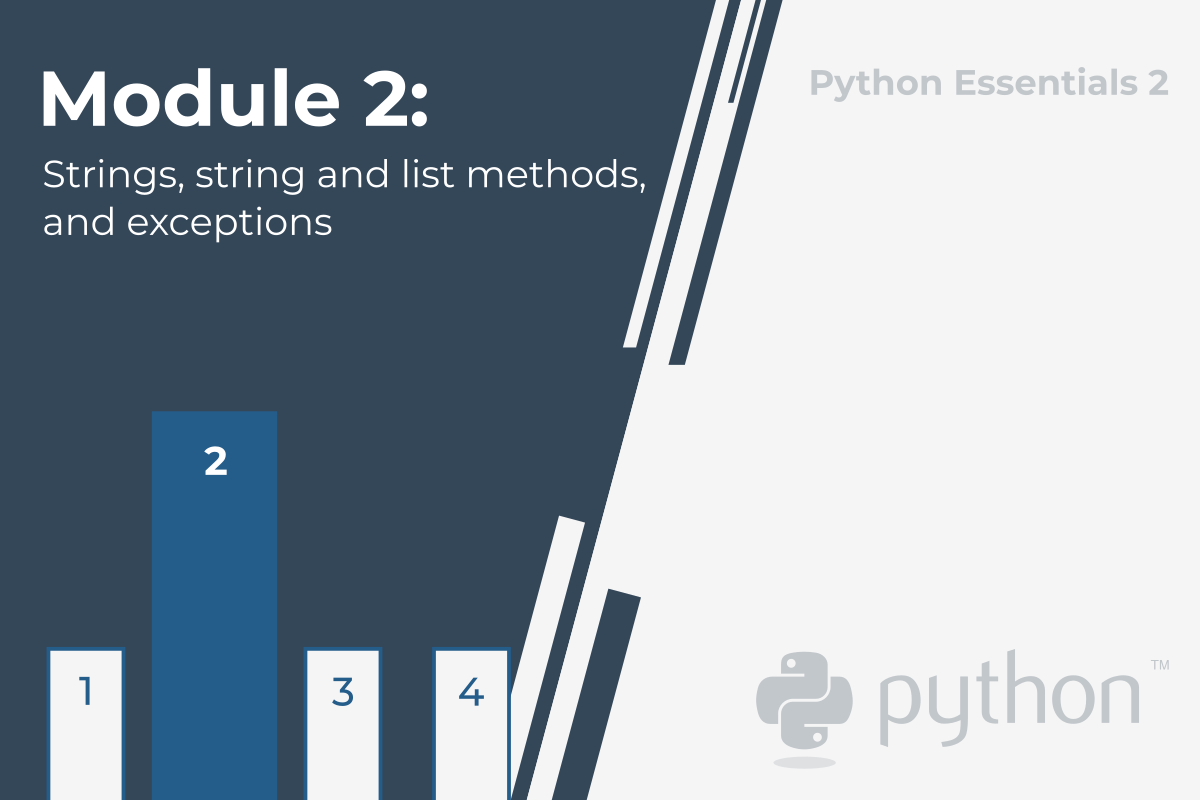
**Python Essentials 2:  
Módulo 2**

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

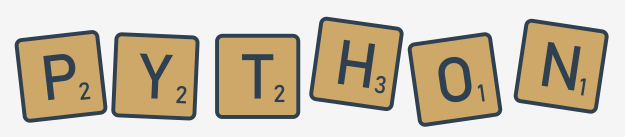
Neste módulo, aprenderá sobre:

* 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.

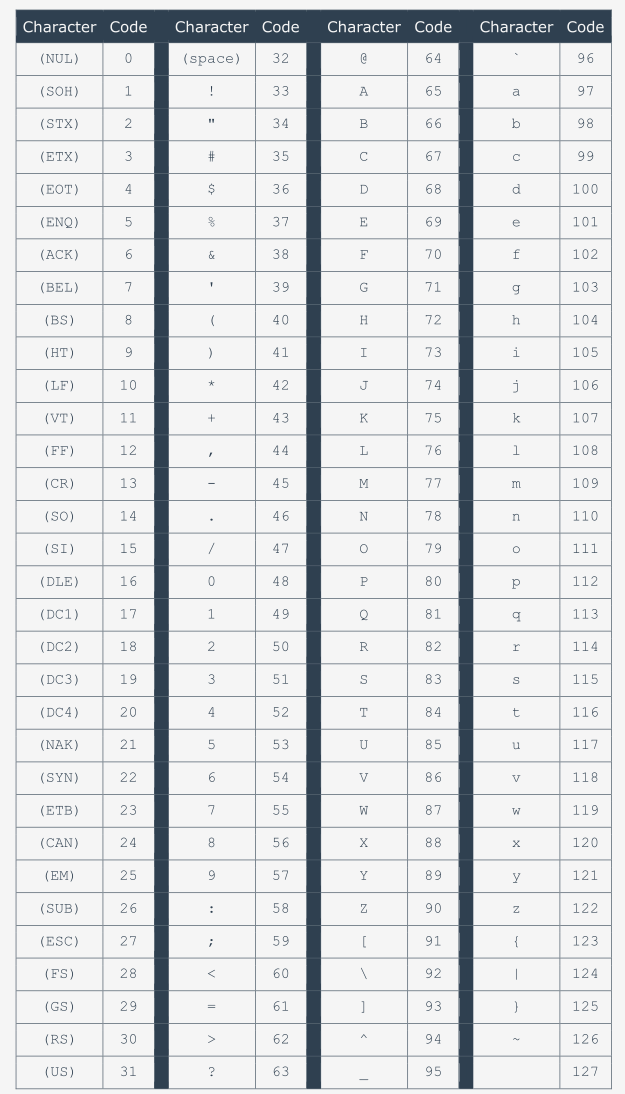
Um exemplo de um whitespace que é completamente invisível a olho nu é um código especial, ou um par de códigos (sistemas operativos diferentes podem tratar esta questão de forma diferente), que são utilizados para marcar as extremidades das linhas dentro dos ficheiros de texto.

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.

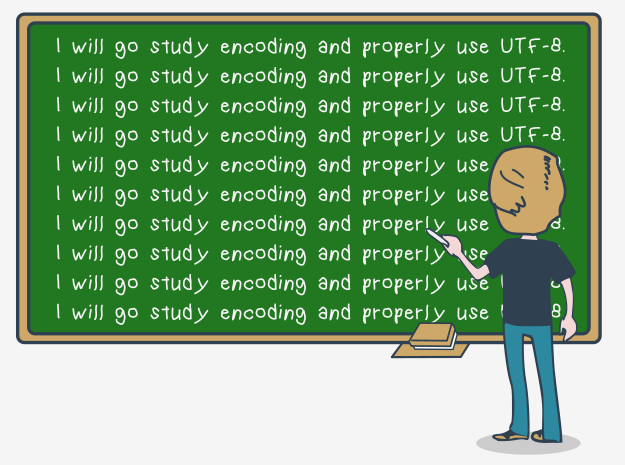
Como pode ver, o UCS-4 é uma norma bastante esbanjadora - aumenta o tamanho de um texto em quatro vezes em comparação com a norma ASCII. Felizmente, existem formas mais inteligentes de codificar textos Unicode.

## 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']

[]

**Comparar strings**

As strings de Python **podem ser comparadas utilizando o mesmo conjunto de operadores** que os utilizados em relação aos números.

Dê uma vista de olhos a estes operadores - todos eles também podem comparar strings:

* ==
* !=
* >
* >=
* <
* <=

Existe um "mas" - os resultados de tais comparações podem por vezes ser um pouco surpreendentes. Não se esqueça que Python não está ciente (não pode de forma alguma) de questões linguísticas sutis - apenas **compara valores de code point, caratere por caratere**.

Os resultados que se obtêm com tal operação são por vezes surpreendentes. Vamos começar com os casos mais simples.

Duas strings são iguais quando consistem nos mesmos carateres, na mesma ordem. Pela mesma forma, duas strings não são iguais quando não consistem nos mesmos carateres, na mesma ordem.

Ambas as comparações dão True como resultado:

'alpha' == 'alpha'

'alpha' != 'Alpha'

A relação final entre strings é determinada pela **comparação do primeiro caratere diferente em ambas as strings** (ter sempre em mente os code points ASCII/UNICODE).

Quando se comparam duas strings de comprimentos diferentes, e a mais curta é idêntica à mais longa, **a string mais longa é considerada maior**.

Tal como aqui:

'alpha' < 'alphabet'

A relação é True.

A comparação de strings é sempre sensível às maiúsculas e minúsculas (**letras maiúsculas são tomadas como menores do que as minúsculas**).

A expressão é True:

'beta' > 'Beta'

**Mesmo que uma string contenha apenas dígitos, ainda assim não é um número**. É interpretada como qualquer outra string regular, e o seu (potencial) aspeto numérico não é tomado em consideração de forma alguma.

Veja os exemplos:

'10' == '010'

'10' > '010'

'10' > '8'

'20' < '8'

'20' < '80'

Eles produzem os seguintes resultados:

False

True

False

True

True

**output**

**Comparar strings com números é geralmente uma má ideia.**

As únicas comparações que pode efetuar com impunidade são estas simbolizadas pelos == e != . O primeiro dá sempre False, enquanto o último produz sempre True.

A utilização de qualquer um dos restantes operadores de comparação irá levantar uma TypeError exceção.

Verifiquemos:

'10' == 10

'10' != 10

'10' == 1

'10' != 1

'10' > 10

Os resultados neste caso são:

False

True

False

True

TypeError exception

**output**

Execute todos os exemplos, e realize mais experiências.

# Sorting

A comparação está estreitamente relacionada com o sorting (ordenação) (ou melhor, o sorting é de facto um caso muito sofisticado de comparação).

Esta é uma boa oportunidade para lhe mostrar duas formas possíveis de **ordenar listas contendo strings**. Tal operação é muito comum no mundo real - sempre que se vê uma lista de nomes, bens, títulos, ou cidades, espera-se que sejam ordenados.

Vamos supor que quer ordenar a seguinte lista:

greek = ['omega', 'alpha', 'pi', 'gamma']

Em geral, o Python oferece duas formas diferentes de ordenar listas.

O primeiro é implementado como **uma função chamada**sorted().

A função toma um argumento (uma lista) e **devolve uma nova lista**, preenchida com os elementos do argumento ordenados. (Nota: esta descrição é um pouco simplificada em comparação com a implementação real - discuti-la-emos mais tarde).

A lista original permanece intacta.

Veja o código no editor, e execute-o. O snippet produz o seguinte output:

['omega', 'alpha', 'pi', 'gamma']

['alpha', 'gamma', 'omega', 'pi']

**output**

O segundo método afeta a própria lista - **nenhuma nova lista é criada**. A ordenação é realizada in situ pelo método chamado sort().

O output não foi alterado:

['omega', 'alpha', 'pi', 'gamma']

['alpha', 'gamma', 'omega', 'pi']

**output**

Se precisar de uma ordem que não seja não-decrescente, tem de convencer a função/método a alterar os seus comportamentos padrão. Vamos discutir isso em breve.

# Strings vs. números

Há duas questões adicionais que devem ser discutidas aqui: **como converter um número (um número inteiro ou um float) numa string, e vice-versa**. Pode ser necessário realizar tal transformação. Além disso, é uma forma rotineira de processar dados de input/output.

A conversão número-string é simples, pois é sempre possível. É feita por uma função chamada str().

Tal como aqui:

itg = 13

flt = 1.3

si = str(itg)

sf = str(flt)

print(si + ' ' + sf)

Output do código:

13 1.3

**output**

A transformação inversa (string-número) é possível quando e só quando a string representa um número válido. Se a condição não for cumprida, espere uma ValueError .

Utilize a função int() , se quiser obter um número inteiro, e float() se precisar de um valor de floating-point.

Tal como aqui:

si = '13'

sf = '1.3'

itg = int(si)

flt = float(sf)

print(itg + flt)

Isto é o que verá na consola:

14.3

**output**

Na secção seguinte, vamos mostrar-lhe alguns programas simples que processam strings.

**Key takeaways**

1. Strings podem ser comparadas com strings utilizando operadores de comparação geral, mas compará-las com números não dá nenhum resultado razoável, visto **nenhuma string poder ser igual** a qualquer número. Por exemplo:

* string == number é sempre False;
* string != number é sempre True;
* string >= number **levanta sempre uma exceção**.

2. A ordenação de listas de strings pode ser feita por:

* uma função chamada sorted(), criando uma lista nova e ordenada;
* um método chamado sort(), que classifica a lista *in situ*

3. Um número pode ser convertido numa string usando a função str() .

4. Uma string pode ser convertida num número (embora não todas as strings) usando ou a função int() ou a função float() . A conversão falha se uma string não contiver uma imagem numérica válida (uma exceção é então levantada).

**Exercício 1**

Qual das linhas a seguir descreve uma condição **verdadeira**?

'smith' > 'Smith'

'Smiths' < 'Smith'

'Smith' > '1000'

'11' < '8'

**Exercício 2**

Qual é o output esperado do seguinte código?

s1 = 'Where are the snows of yesteryear?'

s2 = s1.split()

s3 = sorted(s2)

print(s3[1])

**Exercício 3**

Qual é o resultado esperado do seguinte código?

s1 = '12.8'

i = int(s1)

s2 = str(i)

f = float(s2)

print(s1 == s2)

## Objetivos

* melhorar as habilidades do aluno a operar com strings;
* usar strings para representar dados non-text (não-texto).

## Cenário

Certamente já viu um *display de sete segmentos*.

É um dispositivo (por vezes eletrónico, por vezes mecânico) concebido para apresentar um dígito decimal utilizando um subconjunto de sete segmentos. Se ainda não sabe o que é, consulte o seguinte [artigo](https://en.wikipedia.org/wiki/Seven-segment_display) da Wikipédia.

A sua tarefa é a de escrever **um programa que seja capaz de simular o trabalho de um dispositivo de seven-display**, embora vá utilizar LEDs individuais em vez de segmentos.

Cada dígito é construído a partir de 13 LEDs (uns iluminados, outros escuros, é claro) - é assim que o imaginamos:

# ### ### # # ### ### ### ### ### ###

# # # # # # # # # # # # # #

# ### ### ### ### ### # ### ### # #

# # # # # # # # # # # # #

# ### ### # ### ### # ### ### ###

Nota: o número 8 mostra todas as luzes LED acesas.

O seu código tem de *fazer um display* de qualquer número inteiro não negativo inserido pelo utilizador.

Dica: usar uma lista contendo padrões de todos os dez dígitos pode ser muito útil.

## Dados de teste

Input de amostra:

123

Exemplo de output:

# ### ###

# # #

# ### ###

# # #

# ### ###

Input de amostra:

9081726354

Exemplo de output:

### ### ### # ### ### ### ### ### # #

# # # # # # # # # # # # # #

### # # ### # # ### ### ### ### ###

# # # # # # # # # # # # #

### ### ### # # ### ### ### ### #

RESPOSTA:

digits = [ '1111110', # 0

'0110000', # 1

'1101101', # 2

'1111001', # 3

'0110011', # 4

'1011011', # 5

'1011111', # 6

'1110000', # 7

'1111111', # 8

'1111011', # 9

]

def print\_number(num):

global digits

digs = str(num)

lines = [ '' for lin in range(5) ]

for d in digs:

segs = [ [' ',' ',' '] for lin in range(5) ]

ptrn = digits[ord(d) - ord('0')]

if ptrn[0] == '1':

segs[0][0] = segs[0][1] = segs[0][2] = '#'

if ptrn[1] == '1':

segs[0][2] = segs[1][2] = segs[2][2] = '#'

if ptrn[2] == '1':

segs[2][2] = segs[3][2] = segs[4][2] = '#'

if ptrn[3] == '1':

segs[4][0] = segs[4][1] = segs[4][2] = '#'

if ptrn[4] == '1':

segs[2][0] = segs[3][0] = segs[4][0] = '#'

if ptrn[5] == '1':

segs[0][0] = segs[1][0] = segs[2][0] = '#'

if ptrn[6] == '1':

segs[2][0] = segs[2][1] = segs[2][2] = '#'

for lin in range(5):

lines[lin] += ''.join(segs[lin]) + ' '

for lin in lines:

print(lin)

print\_number(int(input("Enter the number you wish to display: ")))

# A Cifra César: encriptar uma mensagem

Vamos mostrar-lhe quatro programas simples, para apresentar alguns aspetos do processamento de string em Python. Eles são propositadamente simples, mas os problemas do lab serão significativamente mais complicados.

O primeiro problema que queremos mostrar chama-se **Caesar cipher** - mais detalhes aqui: <https://en.wikipedia.org/wiki/Caesar_cipher>.

Esta cifra foi (provavelmente) inventada e usada por Caio Júlio César e suas tropas durante as Guerras Gálicas. A ideia é bastante simples - cada letra da mensagem é substituída pela sua consequente mais próxima (*A* torna-se *B*, *B* torna-se *C*, e assim por diante). A única exceção é *Z*, que se torna *A*.

O programa no editor é uma implementação muito simples (mas funcional) do algoritmo.

Escrevemo-lo utilizando os seguintes pressupostos:

* aceita apenas letras latinas (nota: os romanos não usavam espaços em branco nem dígitos)
* todas as letras da mensagem estão em maiúsculas (nota: os romanos conheciam apenas maiúsculas)

Vamos rastrear o código:

* linha 02: pedir ao utilizador para inserir a mensagem aberta (não encriptada), de uma linha;
* linha 03: preparar uma string para uma mensagem encriptada (vazia por agora)
* linha 04: iniciar a iteração através da mensagem;
* linha 05: se o caratere atual não for alfabético...
* linha 06: ...ignorá-lo;
* linha 07: converter a letra em maiúsculas (é preferível fazê-lo cegamente, em vez de verificar se é necessário ou não)
* linha 08: obter o código da letra e incrementá-lo em um;
* linha 09: se o código resultante tiver “deixado” o alfabeto latino (se for maior do que o código *Z*)...
* linha 10: ...alterá-lo para o código *A*;
* linha 11: anexar o caratere recebido ao fim da mensagem encriptada;
* linha 13: imprimir a cifra.

O código, alimentado com esta mensagem:

AVE CAESAR

tem como output:

BWFDBFTBS

**output**

RESPOSTA:

text = input('Digite a frase para encriptar: ')

cypher = ''

for ch in text:

if not ch.isalpha():

continue

ch = ch.upper()

code = ord(ch) + 1

if code > ord('Z'):

code = ord('A')

cypher += chr(code)

print(cypher)

A transformação inversa deve agora ser clara para si - vamos apenas apresentar-lhe o código tal como está, sem quaisquer explicações.

Veja o código no editor. Verifique cuidadosamente se funciona. Use o criptograma do programa anterior.

**O Processador de Números (Try-Except)**

O terceiro programa mostra um método simples que lhe permite introduzir uma linha cheia de números, e processá-los facilmente. Nota: a função de rotina input() , combinada juntamente com as funções int() ou float() , é inadequada para esta finalidade.

O processamento será extremamente fácil - queremos que os números sejam somados.

Veja o código no editor. Vamos analisá-lo.

A utilização da compreensão de lista pode tornar o código mais magro. Pode fazer isso se quiser.

Vamos apresentar a nossa versão:

* linha 03: pedir ao utilizador para introduzir uma linha preenchida com qualquer número de números (os números podem ser floats)
* linha 04: dividir a linha recebendo uma lista de substrings;
* linha 05: iniciar a soma total a zero;
* linha 06: como a conversão de string-float pode levantar uma exceção, o melhor é continuar com a proteção do bloco try-except;
* linha 07: iterar através da lista...
* linha 08: ...e tentar converter todos os seus elementos em números float; se funcionar, aumentar a soma;
* linha 09: tudo está bem até agora, por isso imprimir a soma;
* linha 10: o programa termina aqui no caso de um erro;
* linha 11: imprimir uma mensagem de diagnóstico mostrando ao utilizador o motivo da falha.

O código tem um ponto fraco importante - apresenta um resultado falso quando o utilizador introduz uma linha vazia. Consegue consertá-lo?

# Numbers Processor.

line = input("Enter a line of numbers - separate them with spaces: ")

strings = line.split()

total = 0

try:

for substr in strings:

total += float(substr)

print("The total is:", total)

except:

print(substr, "is not a number.")

# O validador IBAN

O quarto programa implementa (de uma forma ligeiramente simplificada) um algoritmo utilizado pelos bancos europeus para especificar números de conta. A norma denominada **IBAN** (International Bank Account Number) fornece um método simples e bastante fiável de validação dos números de conta contra erros de digitação simples que possam ocorrer durante a reescrita do número, por exemplo, de documentos em papel, como faturas ou contas, para computadores.

Pode encontrar mais detalhes aqui: <https://en.wikipedia.org/wiki/International_Bank_Account_Number>.

Um número de conta compatível com IBAN consiste em:

* um código de país de duas letras retirado da norma ISO 3166-1 (por exemplo, *FR* para a França, *GB* para a Grã-Bretanha, *DE* para a Alemanha, e assim por diante)
* dois dígitos de verificação utilizados para realizar as verificações de validade - testes rápidos e simples, mas não totalmente fiáveis, mostrando se um número é inválido (distorcido por uma gralha) ou se parece ser válido;
* o número real da conta (até 30 carateres alfanuméricos - o comprimento desta parte depende do país)

A norma diz que a validação requer os seguintes passos (de acordo com a Wikipédia):

* (passo 1) Verificar se o comprimento total do IBAN está correto de acordo com o país (este programa não o fará, mas pode modificar o código para cumprir este requisito, assim o desejar; nota: tem de ensinar ao código todos os comprimentos utilizados na Europa)
* (passo 2) Mover os quatro carateres iniciais para o final da string (ou seja, o código do país e os dígitos de verificação)
* (passo 3) Substituir cada letra na string por dois dígitos, expandindo assim a string, onde *A = 10*, *B = 11*... *Z = 35*;
* (passo 4) Interpretar a string como um número inteiro decimal e calcular o resto desse número na divisão por 97; Se o resto for 1, o teste de verificação de dígito é passado e o IBAN pode ser válido.

Veja o código no editor. Vamos analisá-lo:

* linha 03: pedir ao utilizador para introduzir o IBAN (o número pode conter espaços, uma vez que melhoram significativamente a legibilidade do número...
* linha 04: ...mas remova-os imediatamente)
* linha 05: o IBAN inserido deve consistir apenas em dígitos e letras - se não consistir...
* linha 06:... fazer output da mensagem;
* linha 07: o IBAN não deve ser menor do que 15 carateres (esta é a variante mais curta, usada na Noruega)
* linha 08: se for mais curto, o utilizador é informado;
* linha 09: além disso, o IBAN não pode ter mais de 31 carateres (esta é a variante mais longa, usada em Malta)
* linha 10: se for mais longo, fazer um aviso;
* linha 11: iniciar o processamento em si;
* linha 12: mover os quatro carateres iniciais para o final do número e converter todas as letras em maiúsculas (passo 02 do algoritmo)
* linha 13: esta é a variável utilizada para completar o número, criada através da substituição das letras por dígitos (de acordo com o passo 03 do algoritmo)
* linha 14: iterar através do IBAN;
* linha 15: se o caratere é um dígito...
* linha 16: basta copiá-lo;
* linha 17: caso contrário...
* linha 18: ...convertê-lo em dois dígitos (observe como tal é feito aqui)
* linha 19: a forma convertida do IBAN está pronta - fazer um inteiro dele;
* linha 20: é o resto da divisão de iban2 por 97 igual a 1?
* linha 21: Se sim, então sucesso;
* linha 22: Caso contrário...
* linha 23: ...o número é inválido.

Vamos adicionar alguns dados de teste (todos estes números são válidos - pode invalidá-los alterando qualquer caratere).

* Britânico: GB72 HBZU 7006 7212 1253 00
* Francês: FR76 30003 03620 00020216907 50
* Alemão: DE02100100100152517108

Se for um residente da UE, pode usar o seu próprio número de conta para testes.

# IBAN Validator.

iban = input("Enter IBAN, please: ")

iban = iban.replace(' ','')

if not iban.isalnum():

print("You have entered invalid characters.")

elif len(iban) < 15:

print("IBAN entered is too short.")

elif len(iban) > 31:

print("IBAN entered is too long.")

else:

iban = (iban[4:] + iban[0:4]).upper()

iban2 = ''

for ch in iban:

if ch.isdigit():

iban2 += ch

else:

iban2 += str(10 + ord(ch) - ord('A'))

iban = int(iban2)

if iban % 97 == 1:

print("IBAN entered is valid.")

else:

print("IBAN entered is invalid.")

# Key takeaways

1. As strings são ferramentas chave no processamento moderno de dados, uma vez que a maioria dos dados úteis são na realidade strings. Por exemplo, usar um motor de busca web (que parece bastante trivial nos dias de hoje) utiliza processamento de strings extremamente complexo e complicado, envolvendo quantidades inimagináveis de dados.

2. A comparação de strings de uma forma rigorosa (como faz o Python) pode ser muito insatisfatória quando se trata de pesquisas avançadas (por exemplo, durante extensas consultas a bases de dados). Em resposta a esta exigência, foram criados e implementados vários algoritmos de comparação de strings *fuzzy* (difusas). Estes algoritmos são capazes de encontrar strings que não são iguais no sentido Python, mas são **semelhantes**.

Um desses conceitos é a **distância Hamming**, que é usada para determinar a semelhança de duas strings. Se este problema lhe interessa, pode encontrar mais informações aqui: <https://en.wikipedia.org/wiki/Hamming_distance>. Outra solução do mesmo tipo, mas baseada numa suposição diferente, é a **distância Levenshtein** descrita aqui: <https://en.wikipedia.org/wiki/Levenshtein_distance>.

3. Outra forma de comparar strings é encontrar a sua semelhança *acústica*, o que significa um processo que leva a determinar se duas strings soam de forma semelhante (como "raise" e "race"). Essa semelhança deve ser estabelecida para cada língua (ou mesmo dialeto) separadamente.

Um algoritmo utilizado para efetuar tal comparação para a língua inglesa chama-se **Soundex** e foi inventado - não vai acreditar - em 1918. Pode descobrir mais informações aqui: <https://en.wikipedia.org/wiki/Soundex>.

4. Devido à limitada precisão de dados nativos float e inteiros, é por vezes razoável armazenar e processar valores numéricos enormes como strings. Esta é a técnica que o Python utiliza quando se o força a funcionar com um número inteiro constituído por um número muito grande de dígitos.

# LAB Melhorar a cifra de César

Difícil

## Pré-requisitos

Módulo 1.11.1.1, Módulo 1.11.1.2

## Objetivos

* melhorar as habilidades do aluno a operar com strings;
* converter carateres em código ASCII e vice-versa.

## Cenário

Já está familiarizado com a cifra César, e é por isso que queremos que melhore o código que lhe mostrámos recentemente.

A cifra original de César desloca cada caratere por um: *a* torna-se *b*, *z* torna-se *a*, e assim por diante. Vamos torná-la um pouco mais difícil, e permitir que o valor deslocado venha do intervalo 1..25 inclusive.

Além disso, deixe que o código preserve as letras (letras minúsculas permanecerão minúsculas) e todos os carateres não alfabéticos devem permanecer intocados.

A sua tarefa é escrever um programa que:

* pede ao utilizar uma linha de texto para encriptar;
* pede ao utilizador um valor de deslocamento (um número inteiro no intervalo 1..25 - nota: deve forçar o utilizador a inserir um valor de deslocamento válido (não desista e não deixe que maus dados o enganem!);
* imprime o texto codificado.

Teste o seu código utilizando os dados por nós fornecidos.

## Dados de teste

Input de amostra:

abcxyzABCxyz 123

2

Exemplo de output:

cdezabCDEzab 123

Input de amostra:

The die is cast

25

Exemplo de output:

Sgd chd hr bzrs

# LAB Palíndromos

Fácil

## Objetivos

* melhorar as habilidades do aluno a operar com strings;
* encorajar o estudante a procurar soluções não óbvias.

## Cenário

Sabe o que é um palíndromo?

É uma palavra que parece a mesma quando lida para a frente e para trás. Por exemplo, "rever" é um palíndromo, enquanto "leal" não é.

A sua tarefa é escrever um programa que:

* pede ao utilizador algum texto;
* verifica se o texto introduzido é um palíndromo, e imprime o resultado.

Nota:

* assuma que uma string vazia não é um palíndromo;
* trate as letras maiúsculas e minúsculas como iguais;
* os espaços não são tidos em conta durante a verificação - trate-os como inexistentes;
* existem mais do que algumas soluções corretas - tente encontrar mais do que uma.

Teste o seu código utilizando os dados por nós fornecidos.

## Dados de teste

Input de amostra:

Ten animals I slam in a net

Exemplo de output:

It's a palindrome

Input de amostra:

Eleven animals I slam in a net

Exemplo de output:

It's not a palindrome

# LAB Anagramas

Fácil

## Objetivos

* melhorar as habilidades do aluno a operar com strings;
* converter strings em listas, e vice-versa.

## Cenário

Um anagrama é uma nova palavra formada pelo rearranjo das letras de uma palavra, usando todas as letras originais exatamente uma vez. Por exemplo, as frases “o bom programa” e “agarram o pombo” são anagramas, enquanto “Eu sou” e “Você é” não são.

A sua tarefa é escrever um programa que:

* pede ao utilizador dois textos separados;
* verifica se os textos introduzidos são anagramas e imprime o resultado.

Nota:

* assuma que duas strings vazias não são anagramas;
* trate as letras maiúsculas e minúsculas como iguais;
* os espaços não são tidos em conta durante a verificação - trate-os como inexistentes

Teste o seu código utilizando os dados por nós fornecidos.

## Dados de teste

Input de amostra:

Listen

Silent

Exemplo de output:

Anagrams

Input de amostra:

modern

norman

Exemplo de output:

Not anagrams

# LAB O digito da vida

Fácil

## Objetivos

* melhorar as habilidades do aluno a operar com strings;
* converter inteiros em strings, e vice-versa.

## Cenário

Alguns dizem que o *Dígito da Vida* é um dígito avaliado usando o aniversário de alguém. É simples - basta somar todos os dígitos da data. Se o resultado contiver mais do que um dígito, terá de repetir a adição até obter exatamente um dígito. Por exemplo:

* 1 de janeiro de 2017 = 2017 01 01
* 2 + 0 + 1 + 7 + 0 + 1 + 0 + 1 = 12
* 1 + 2 = 3

*3* é o dígito que procurámos e encontrámos.

A sua tarefa é escrever um programa que:

* pergunta ao utilizador o seu aniversário (no formato AAAAMMDD, ou AAAADDMM, ou MMDDAAAA - na verdade, a ordem dos dígitos não importa)
* faz output do *Dígito de Vida* para a data.

Teste o seu código utilizando os dados por nós fornecidos.

## Dados de teste

Input de amostra:

19991229

Exemplo de output:

6

Input de amostra:

20000101

Exemplo de output:

4

# LAB Encontre uma palavra

Médio

## Objetivos

* melhorar as habilidades do aluno a operar com strings;
* utilizar o método find() para pesquisar strings.

## Cenário

Vamos jogar um jogo. Dar-lhe-emos duas strings: uma sendo uma palavra (por exemplo, “dog”) e a segunda sendo uma combinação de quaisquer carateres.

A sua tarefa é a de escrever um programa que responda à seguinte questão: **os carateres que compõem a primeira string estão escondidos dentro da segunda string?**

Por exemplo:

* se a segunda string for dada como “vcxzxduybfdsobywuefgas”, a resposta é yes;
* se a segunda string for “vcxzxdcybfdstbywuefsas”, a resposta é no (visto não haver as letras “d”, “o”, ou “g”, nesta ordem)

Dicas:

* deve utilizar as variantes de dois argumentos das funções pos() dentro do seu código;
* não se preocupe com a sensibilidade a maiúsculas e minúsculas.

Teste o seu código utilizando os dados por nós fornecidos.

## Dados de teste

Input de amostra:

donor

Nabucodonosor

Exemplo de output:

Yes

Input de amostra:

donut

Nabucodonosor

Exemplo de output:

No

# LAB Sudoku

Difícil

## Objetivos

* melhorar as competências do aluno em operar com strings e listas;
* converter strings em listas.

## Cenário

Como provavelmente sabe, o *Sudoku* é um puzzle de colocação de números jogado num tabuleiro 9x9. O jogador tem de preencher o tabuleiro de uma forma muito específica:

* cada linha do tabuleiro deve conter todos os dígitos de 1 a 9 (a ordem não importa)
* cada coluna do tabuleiro deve conter todos os dígitos de 1 a 9 (novamente, a ordem não importa)
* cada uma das nove "regiôes" 3x3 (vamos nomeá-las “sub-quadrados”) da tabela deve conter todos os dígitos de 1 a 9.

Se precisar de mais detalhes, pode encontrá-los [aqui](https://en.wikipedia.org/wiki/Sudoku).

A sua tarefa é escrever um programa que:

* leia 9 linhas do Sudoku, cada uma contendo 9 dígitos (verifique cuidadosamente se os dados introduzidos são válidos)
* faz o output Yes se o Sudoku for válido, e No caso contrário.

Teste o seu código utilizando os dados por nós fornecidos.

## Dados de teste

Input de amostra:

295743861

431865927

876192543

387459216

612387495

549216738

763524189

928671354

154938672

Exemplo de output:

Yes

Input de amostra:

195743862

431865927

876192543

387459216

612387495

549216738

763524189

928671354

254938671

Exemplo de output:

No

# LAB PASTES

## MELHORAR A CIFRA DE CESAR

# input text to encrypt

text = input("Enter message: ")

# enter valid shift value (repeat until it succeeds)

shift = 0

while shift == 0:

try:

shift = int(input("Enter cipher's shift (1..25): "))

if shift not in range(1,26):

raise ValueError

except ValueError:

shift = 0

if shift == 0:

print("Bad shift value!")

cipher = ''

for char in text:

# is it a letter?

if char.isalpha():

# shift its code

code = ord(char) + shift

# find the code of the first letter (upper- or lower-case)

if char.isupper():

first = ord('A')

else:

first = ord('a')

# make correction

code -= first

code %= 26

# append encoded character to message

cipher += chr(first + code)

else:

# append original character to message

cipher += char

print(cipher)

## PALÍNDROMOS

text = input("Enter text: ")

# remove all spaces...

text = text.replace(' ','')

# ... and check if the word is equal to reversed itself

if len(text) > 1 and text.upper() == text[::-1].upper():

print("It's a palindrome")

else:

print("It's not a palindrome")

## ANAGRAMA

str\_1 = input("Enter the first string: ")

str\_2 = input("Enter the second string: ")

strx\_1 = ''.join(sorted(list(str\_1.upper().replace(' ',''))))

strx\_2 = ''.join(sorted(list(str\_2.upper().replace(' ',''))))

if len(strx\_1) > 0 and strx\_1 == strx\_2:

print("Anagrams")

else:

print("Not anagrams")

## O DIGITO DA VIDA

date = input("Enter your birthday date (in the following format: YYYYMMDD or YYYYDDMM, 8 digits): ")

if len(date) != 8 or not date.isdigit():

print("Invalid date format.")

else:

while len(date) > 1:

the\_sum = 0

for dig in date:

the\_sum += int(dig)

print(date)

date = str(the\_sum)

print("Your Digit of Life is: " + date)

## ENCONTRE UMA PALAVRA

word = input("Enter the word you wish to find: ").upper()

strn = input("Enter the string you wish to search through: ").upper()

found = True

start = 0

for ch in word:

pos = strn.find(ch, start)

if pos < 0:

found = False

break

start = pos + 1

if found:

print("Yes")

else:

print("No")

## SUDOKU

# A function that checks whether a list passed as an argument contains

# nine digits from '1' to '9'.

def checkset(digs):

return sorted(list(digs)) == [chr(x + ord('0')) for x in range(1, 10)]

# A list of rows representing the sudoku.

rows = [ ]

for r in range(9):

ok = False

while not ok:

row = input("Enter row #" + str(r + 1) + ": ")

ok = len(row) == 9 or row.isdigit()

if not ok:

print("Incorrect row data - 9 digits required")

rows.append(row)

ok = True

# Check if all rows are good.

for r in range(9):

if not checkset(rows[r]):

ok = False

break

# Check if all columns are good.

if ok:

for c in range(9):

col = []

for r in range(9):

col.append(rows[r][c])

if not checkset(col):

ok = False

break

# Check if all sub-squares (3x3) are good.

if ok:

for r in range(0, 9, 3):

for c in range(0, 9, 3):

sqr = ''

# Make a string containing all digits from a sub-square.

for i in range(3):

sqr += rows[r+i][c:c+3]

if not checkset(list(sqr)):

ok = False

break

# Print the final verdict.

if ok:

print("Yes")

else:

print("No")

**Erros, fracassos e outras pragas**

**Qualquer coisa que possa correr mal, irá correr mal**.

Esta é a lei de Murphy, e funciona em todo o lado e sempre. A execução do seu código também pode correr mal. Se puder, irá.

import math

x = float(input("Enter x: "))

y = math.sqrt(x)

print("The square root of", x, "equals to", y)

Veja o código no editor. Há pelo menos duas formas possíveis de "correr mal". Consegue vê-las?

* Como um utilizador é capaz de introduzir uma string completamente arbitrária, **não há garantia de que a string possa ser convertida num valor float** - esta é a primeira vulnerabilidade do código;
* a segundo é que a função sqrt()**falha se obtiver um argumento negativo**.

Poderá receber uma das seguintes mensagens de erro.

Algo assim:

Enter x: Abracadabra

Traceback (most recent call last):

File "sqrt.py", line 3, in <module>

x = float(input("Enter x: "))

ValueError: could not convert string to float: 'Abracadabra'

**output**

Ou algo assim:

Enter x: -1

Traceback (most recent call last):

File "sqrt.py", line 4, in <module>

y = math.sqrt(x)

ValueError: math domain error

**output**

Pode proteger-se de tais surpresas? Claro que pode. Além disso, tem de o fazer para ser considerado um bom programador.

**Exceções**

Cada vez que o seu código tenta fazer algo de errado/tolo/irresponsável/maluco/imprevisível, o Python faz duas coisas:

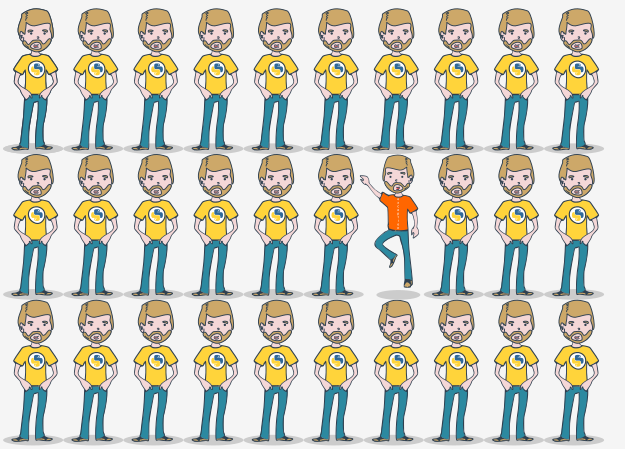
* **para o seu programa**;
* cria um tipo especial de dados, chamado uma **exceção**.

Ambas estas atividades são designadas por **levantar uma exceção**. Podemos dizer que o Python levanta sempre uma exceção (ou que **foi levantada uma exceção**) quando não tem ideia do que fazer com o seu código.

O que acontece a seguir?

* a exceção levantada espera que alguém ou alguma coisa a note e se encarregue dela;
* se nada acontecer para tratar da exceção levantada, o programa será **terminado à força**, e verá uma **mensagem de erro** enviada para a consola pelo Python;
* caso contrário, se a exceção for cuidada e **tratada** adequadamente, o programa suspenso pode ser retomado e a sua execução pode continuar.

O Python fornece ferramentas eficazes que lhe permitem **observar exceções, identificá-las e tratá-las** eficientemente. Isto é possível devido ao facto de todas as potenciais exceções terem os seus nomes inequívocos, pelo que se pode categorizá-los e reagir adequadamente.



Já conhece alguns nomes de exceção. Dê uma vista de olhos à seguinte mensagem de diagnóstico:

ValueError: math domain error

**output**

A palavra destacada acima é apenas o **nome da exceção**. Vamos familiarizar-nos com algumas outras exceções.

Veja o código no editor. Execute o programa (obviamente incorreto).

value = 1

value /= 0

Verá a seguinte mensagem em resposta:

Traceback (most recent call last):

File "div.py", line 2, in

value /= 0

ZeroDivisionError: division by zero

**output**

Este erro de exceção é chamado **ZeroDivisionError**.

Veja o código no editor. O que acontecerá quando o executar? Verifique.

my\_list = []

x = my\_list[0]

Verá a seguinte mensagem em resposta:

Traceback (most recent call last):

File "lst.py", line 2, in

x = list[0]

IndexError: list index out of range

**output**

Este é o **IndexError**.

Como se **tratam** (em inglês, handle) as exceções? A palavra try é fundamental para a solução.

Além disso, também é uma keyword.

A receita para o sucesso é a seguinte:

* primeiro, é preciso ***tentar* fazer alguma coisa**;
* a seguir, tem de **verificar se tudo correu bem**.

Mas não seria melhor verificar primeiro todas as circunstâncias, e depois fazer algo apenas se for seguro?

Assim como o exemplo no editor.

first\_number = int(input("Enter the first number: "))

second\_number = int(input("Enter the second number: "))

if second\_number != 0:

print(first\_number / second\_number)

else:

print("This operation cannot be done.")

print("THE END.")

Assumidamente, esta forma pode parecer a mais natural e compreensível, mas na realidade este método não torna a programação mais fácil. Todas estas verificações podem tornar **o seu código inchado e ilegível**.

O Python prefere uma abordagem completamente diferente.

Veja o código no editor. Esta é a abordagem favorita do Python.

first\_number = int(input("Enter the first number: "))

second\_number = int(input("Enter the second number: "))

try:

print(first\_number / second\_number)

except:

print("This operation cannot be done.")

print("THE END.")

Nota:

* a keyword try **inicia um bloco do código** que pode ou não estar a funcionar corretamente;
* a seguir, o Python tenta realizar a ação arriscada; se falhar, é levantada uma exceção e o Python começa a procurar por uma solução;
* a keyword except inicia um pedaço de código que será **executado se alguma coisa dentro do bloco**try**correr mal** - se uma exceção é levantada dentro de um bloco try anterior, **falhará aqui**, pelo que o código localizado após a keyword except deverá fornecer uma **reação adequada** à exceção levantada;
* retornar ao nível de nesting anterior termina a secção do **try-except**.

Execute o código e teste o seu comportamento.

Vamos resumir isto:

try:

:

:

except:

:

:

* no primeiro passo, o Python tenta executar todas as instruções colocadas entre as try: e except: declarações;
* se nada estiver errado com a execução e todas as instruções forem executadas com sucesso, a execução salta para o ponto após a última linha do bloco except: , e a execução do bloco é considerada completa;
* se alguma coisa correr mal dentro do bloco try: e except: , a execução salta imediatamente para fora do bloco e para a primeira instrução localizada após a keyword except: ; isto significa que algumas das instruções do bloco podem ser silenciosamente omitidas.

Veja o código no editor. Ajudá-lo-á a compreender este mecanismo.

try:

print("1")

x = 1 / 0

print("2")

except:

print("Oh dear, something went wrong...")

print("3")

Esta é o output que produz:

1

Oh dear, something went wrong...

3

**output**

Nota: a instrução print("2") foi perdida no processo.

Esta abordagem tem uma desvantagem importante - se houver a possibilidade de mais do que uma exceção poder saltar para um ramo except: , poderá ter **dificuldade em descobrir o que realmente aconteceu**.

Tal como no nosso código no editor. Execute-o e veja o que acontece.

try:

x = int(input("Enter a number: "))

y = 1 / x

except:

print("Oh dear, something went wrong...")

print("THE END.")

A mensagem: Oh dear, something went wrong... que aparece na consola nada diz sobre a razão, enquanto que existem duas causas possíveis para a exceção:

* dados não inteiros inseridos pelo utilizador;
* um valor inteiro igual a 0 atribuído à variável x .

Tecnicamente, existem duas maneiras de resolver o problema:

* construir dois blocos consecutivos try-except , um para cada possível motivo de exceção (fácil, mas causará um crescimento do código desfavorável)
* utilizar uma variante mais avançada da instrução.

É assim que se parece:

try:

:

except exc1:

:

except exc2:

:

except:

:

É assim que funciona:

* se o ramo try levanta a exceção exc1 , será tratado pelo bloco except exc1: ;
* da mesma forma, se o ramo try levanta a exceção exc2 , será tratado pelo bloco except exc2: ;
* se o ramo try levanta qualquer outra exceção, será tratado pelo bloco não nomeado except .

Passemos à próxima parte do curso e vejamo-lo em ação.

Veja o código no editor. A nossa solução está lá.

try:

x = int(input("Enter a number: "))

y = 1 / x

print(y)

except ZeroDivisionError:

print("You cannot divide by zero, sorry.")

except ValueError:

print("You must enter an integer value.")

except:

print("Oh dear, something went wrong...")

print("THE END.")

O código, quando executado, produz uma das quatro variantes de output seguintes:

* se introduzir um valor inteiro válido, não nulo (por exemplo 5) diz:  
    
  0.2
* THE END.

**output**

* se introduzir 0, diz:  
    
  You cannot divide by zero, sorry.
* THE END.

**output**

* se introduzir qualquer string não inteira, verá:  
    
  You must enter an integer value.
* THE END.

**output**

* (localmente na sua máquina) se premir Ctrl-C enquanto o programa aguarda a entrada do utilizador (o que provoca uma exceção chamada KeyboardInterrupt), o programa diz:  
    
  Oh dear, something went wrong...
* THE END.

**output**

Não se esqueça:

* os ramos except são pesquisados na mesma ordem em que aparecem no código;
* não deve utilizar mais do que um, exceto um ramo com um certo nome de exceção;
* o número de diferentes ramos except é arbitrário - a única condição é que se utilizar try, deve colocar pelo menos um except (nomeado ou não) depois dele;
* a keyword except não deve ser utilizada sem um precedente try;
* se algum dos ramos except for executado, nenhum outro ramo será visitado;
* se nenhum dos ramos except especificados correspondem à exceção levantada, a exceção permanece sem ser tratada (discutiremos isso em breve)
* se um não nomeado except ramo existir (um sem nome de exceção), tem de ser especificado como o último.

try:

:

except exc1:

:

except exc2:

:

except:

:

Vamos continuar as experiências agora.

Veja o código no editor. Modificámos o programa anterior - removemos o ramo ZeroDivisionError .

try:

x = int(input("Enter a number: "))

y = 1 / x

print(y)

except ValueError:

print("You must enter an integer value.")

except:

print("Oh dear, something went wrong...")

print("THE END.")

O que acontece agora se o utilizador introduzir 0 como um input?

Como **não há ramos dedicados** para divisão por zero, a exceção levantada cai no **ramo geral (sem nome)**; isto significa que, neste caso, o programa dirá:

Oh dear, something went wrong...

THE END.

**output**

Experimente você mesmo. Execute o programa.

Vamos estragar o código mais uma vez.

Olhe para o programa no editor. Desta vez, retirámos o ramo não nomeado.

try:

x = int(input("Enter a number: "))

y = 1 / x

print(y)

except ValueError:

print("You must enter an integer value.")

print("THE END.")

O utilizador insere 0 mais uma vez, e:

* a exceção levantada não será tratada por ValueError - não tem nada a ver com isso;
* uma vez que não há outro ramo, deve ver esta mensagem:  
    
  Traceback (most recent call last):
* File "exc.py", line 3, in
* y = 1 / x
* ZeroDivisionError: division by zero

Aprendeu bastante sobre o tratamento de exceções em Python. Na secção seguinte, centrar-nos-emos nas exceções incorporadas em Python e nas suas hierarquias.

# Key takeaways

1. Uma exceção é um evento na execução de um programa causado por uma situação anormal. A exceção deve ser tratada para evitar a terminação do programa. A parte do seu código que é suspeita de ser a origem da exceção deve ser colocada dentro do ramo try .

Quando a exceção acontece, a execução do código não é terminada, mas em vez disso salta para o ramo except . Este é o local onde deve ter lugar o tratamento da exceção. O esquema geral de tal construção parece-se com o seguinte:

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except:

:

# Crisis management takes place here.

:

:

# Back to normal.

:

2. Se precisar de tratar de mais do que uma exceção proveniente do mesmo ramo try , pode adicionar mais do que um ramo except , mas é preciso rotulá-los com nomes de exceção diferentes, como este:

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except Except\_1:

# Crisis management takes place here.

except Except\_2:

# We save the world here.

:

# Back to normal.

:

No máximo, um dos ramos except é executado - nenhum dos ramos é executado quando a exceção levantada não coincide com as exceções especificadas.

3. Não pode acrescentar mais do que um ramo anónimo (não nomeado) except após os nomeados.

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except Except\_1:

# Crisis management takes place here.

except Except\_2:

# We save the world here.

except:

# All other issues fall here.

:

# Back to normal.

:

**Exercício 1**

Qual é o output esperado do seguinte código?

try:

print("Let's try to do this")

print("#"[2])

print("We succeeded!")

except:

print("We failed")

print("We're done")

**Exercício 2**

Qual é o output esperado do seguinte código?

try:

print("alpha"[1/0])

except ZeroDivisionError:

print("zero")

except IndexingError:

print("index")

except:

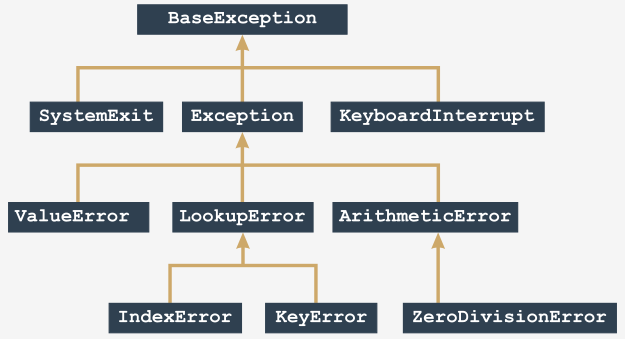
print("some")

**Exceções II**

O Python 3 define **63 exceções incorporadas**, e todas elas formam uma **hierarquia em forma de árvore**, embora a árvore seja um pouco estranha, uma vez que a sua raiz está localizada no topo.

Algumas das exceções incorporadas são mais gerais (incluem outras exceções) enquanto outras são completamente concretas (representam-se apenas a si próprias). Podemos dizer que **quanto mais próxima da raiz se encontra uma exceção, mais geral (abstrata) ela é**. Por sua vez, as exceções localizadas nas extremidades dos ramos (podemos chamar-lhes **folhas**) são concretas.

Dê uma vista de olhos à figura:



Mostra uma pequena secção da árvore de exceção completa. Vamos começar a examinar a árvore a partir da ZeroDivisionError folha.

Nota:

* ZeroDivisionError é um caso especial de uma classe de exceção mais geral chamada ArithmeticError;
* ArithmeticError é um caso especial de uma classe de exceção mais geral chamada apenas Exception;
* Exception é um caso especial de uma classe mais geral chamada BaseException;

Podemos descrevê-la da seguinte forma (note-se a direção das setas - apontam sempre para a entidade mais geral):

BaseException  
↑  
Exception  
↑  
ArithmeticError  
↑  
ZeroDivisionError

Vamos mostrar-lhe como funciona esta generalização. Vamos começar com um código realmente simples.

Veja o código no editor. É um exemplo simples para começar. Execute-o.

try:

y = 1 / 0

except ZeroDivisionError:

print("Oooppsss...")

print("THE END.")

O output que esperamos ver é semelhante a este:

Oooppsss...

THE END.

**output**

Agora veja o código abaixo:

try:

y = 1 / 0

except ArithmeticError:

print("Oooppsss...")

print("THE END.")

Alguma coisa mudou nele - nós substituímos ZeroDivisionError por ArithmeticError.

Já sabe que ArithmeticError é uma classe geral, incluindo (entre outros) a exceção ZeroDivisionError .

Assim, o output do código permanece inalterado. Teste-o.

Isto também significa que a substituição do nome da exceção por Exception ou BaseException não vai mudar o comportamento do programa.

Vamos resumir:

* cada exceção levantada **cai no primeiro ramo correspondente**;
* o ramo correspondente não tem de especificar exatamente a mesma exceção - basta que a exceção seja **mais geral** (mais abstrata) do que a exceção levantada.

Veja o código no editor. O que vai acontecer aqui?

try:

y = 1 / 0

except ZeroDivisionError:

print("Zero Division!")

except ArithmeticError:

print("Arithmetic problem!")

print("THE END.")

O primeiro ramo correspondente é o que contém ZeroDivisionError. Isto significa que a consola irá mostrar:

Zero division!

THE END.

**output**

Irá mudar alguma coisa se trocarmos os dois ramos except ? Tal como aqui em baixo:

try:

y = 1 / 0

except ArithmeticError:

print("Arithmetic problem!")

except ZeroDivisionError:

print("Zero Division!")

print("THE END.")

A mudança é radical - o output do código é agora:

Arithmetic problem!

THE END.

**output**

Porquê, se a exceção levantada é a mesma que anteriormente?

A exceção é a mesma, mas a exceção mais geral é agora listada em primeiro lugar - também irá apanhar todas as divisões zero. Significa também que não há hipótese de qualquer exceção atingir o ramo ZeroDivisionError . Este ramo é agora completamente inacessível.

Lembre-se:

* a ordem dos ramos é importante!
* não ponha exceções mais gerais antes de exceções mais concretas;
* isto tornará este último inalcançável e inútil;
* Além disso, tornará o seu código confuso e inconsistente;
* O Python não irá gerar quaisquer mensagens de erro relativamente a esta questão.

Se quiser **tratar duas ou mais exceções** da mesma forma, pode usar a seguinte sintaxe:

try:

:

except (exc1, exc2):

:

É simplesmente necessário colocar todos os nomes de exceção envolvidos numa lista separada por vírgulas, e não esquecer os parêntesis.

Se uma **exceção for levantada dentro de uma função**, pode ser tratada:

* dentro da função;
* fora da função;

Vamos começar com a primeira variante - veja o código no editor.

def bad\_fun(n):

try:

return 1 / n

except (ArithmeticError, ZeroDivisionError):

print("Arithmetic Problem!")

return None

bad\_fun(0)

print("THE END.")

A exceção ZeroDivisionError (sendo um caso concreto da classe de exceção ArithmeticError ) é levantada dentro da função bad\_fun() , e não deixa a função - a própria função cuida dela.

O programa faz o output:

Arithmetic problem!

THE END.

**output**

Também é possível deixar que a exceção se propague **fora da função**. Vamos testá-la agora.

Veja o código em baixo:

def bad\_fun(n):

return 1 / n

try:

bad\_fun(0)

except ArithmeticError:

print("What happened? An exception was raised!")

print("THE END.")

O problema tem de ser resolvido pelo invocador (ou pelo invocador do invocador, e assim por diante).

O programa faz o output:

What happened? An exception was raised!

THE END.

**output**

Nota: a **exceção levantada pode ultrapassar os limites da função e do módulo**, e viajar através da cadeia de invocação procurando uma cláusula except correspondente capaz de a tratar.

Se não existir tal cláusula, a exceção permanece sem ser acionada, e o Python resolve o problema à sua maneira padrão - **terminando o seu código e emitindo uma mensagem de diagnóstico**.

Agora vamos suspender esta discussão, pois queremos apresentar-lhe uma nova instrução Python.

A instrução raise levanta a exceção especificada chamada exc como se fosse levantada de uma forma normal (natural):

raise exc

Nota: raise é uma keyword.

A instrução permite-lhe:

* **simular o levantamento de exceções reais** (por exemplo, para testar a sua estratégia de tratamento)
* parcialmente **tratar uma exceção** e tornar outra parte do código responsável por completar o tratamento (separação de preocupações).

Veja o código no editor. É assim que o pode utilizar na prática.

def bad\_fun(n):

raise ZeroDivisionError

try:

bad\_fun(0)

except ArithmeticError:

print("What happened? An error?")

print("THE END.")

O output do programa permanece inalterado.

Desta forma, pode **testar a sua rotina de tratamento de exceções** sem forçar o código a fazer coisas estúpidas.

A instrução raise também pode ser utilizada da seguinte maneira (observe a ausência do nome da exceção):

raise

Existe uma restrição grave: este tipo de instrução raise pode ser utilizada **dentro do ramo**exceptapenas; usá-la em qualquer outro contexto causa um erro.

A instrução voltará imediatamente a levantar a mesma exceção que é atualmente tratada.

Graças a isto, pode distribuir o tratamento de exceções entre diferentes partes do código.

Veja o código no editor. Execute-o - vamos vê-lo em ação.

def bad\_fun(n):

try:

return n / 0

except:

print("I did it again!")

raise

try:

bad\_fun(0)

except ArithmeticError:

print("I see!")

print("THE END.")

A exceção ZeroDivisionError é levantado duas vezes:

* primeiro, dentro da parte do código try (isto é causado pela divisão real zero)
* segundo, dentro da parte except pela instrução raise .

Com efeito, o código faz o output:

I did it again!

I see!

THE END.

Agora é um bom momento para lhe mostrar outra instrução Python, chamada assert. Esta é uma keyword.

assert expression

Como é que funciona?

* Ela avalia a expressão;
* se a expressão for avaliada para True, ou um valor numérico não nulo, ou uma string não vazia, ou qualquer outro valor diferente de None, não fará mais nada;
* caso contrário, levanta automática e imediatamente uma exceção chamada AssertionError (neste caso, dizemos que a assertion falhou)

Como pode ser utilizada?

* pode querer inseri-la no seu código onde quer estar **absolutamente a salvo de dados evidentemente errados**, e onde não tem a certeza absoluta de que os dados já foram cuidadosamente examinados (por exemplo, dentro de uma função utilizada por outra pessoa)
* levantar uma exceção AssertionError assegura que o seu código não produz resultados inválidos, e mostra claramente a natureza do fracasso;
* **as assertions não substituem as exceções nem validam os dados** - são seus suplementos.

Se as exceções e a validação de dados forem como uma condução cuidadosa, a assertion pode desempenhar o papel de um airbag.

Vamos ver a instrução assert em ação. Veja o código no editor. Execute-o.

import math

x = float(input("Enter a number: "))

assert x >= 0.0

x = math.sqrt(x)

print(x)

O programa corre sem falhas se introduzir um valor numérico válido maior ou igual a zero; caso contrário, ele para e emite a seguinte mensagem:

Traceback (most recent call last):

File ".main.py", line 4, in

assert x >= 0.0

AssertionError

# Key takeaways

1. Não pode acrescentar mais do que um ramo anónimo (não nomeado) except após os nomeados.

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except Except\_1:

# Crisis management takes place here.

except Except\_2:

# We save the world here.

except:

# All other issues fall here.

:

# Back to normal.

:

2. Todas as exceções Python pré-definidas formam uma hierarquia, ou seja, algumas delas são mais gerais (a que se chama BaseException é a mais geral), enquanto outras são mais ou menos concretas (por exemplo, IndexError é mais concreta do que LookupError).

Não deve colocar exceções mais concretas antes das mais gerais dentro da mesma sequência de ramificações except . Por exemplo, pode fazer isto:

try:

# Risky code.

except IndexError:

# Taking care of mistreated lists

except LookupError:

# Dealing with other erroneous lookups

mas não o faça (a menos que tenha a certeza absoluta de que deseja que alguma parte do seu código seja inútil)

try:

# Risky code.

except LookupError:

# Dealing with erroneous lookups

except IndexError:

# You'll never get here

3. A declaração Python raise ExceptionName pode levantar uma exceção sob demanda. A mesma declaração, mas sem *ExceptionName*, pode ser usada dentro do ramo try **apenas**, e levanta a mesma exceção que está atualmente a ser tratada.

4. A declaração Python assert expression avalia a *expressão* e levanta a exceção AssertError quando a *expressão* é igual a zero, uma string vazia ou None. Pode utilizá-la para proteger algumas partes críticas do seu código de dados devastadores.

**Exercício 1**

Qual é o output esperado do seguinte código?

try:

print(1/0)

except ZeroDivisionError:

print("zero")

except ArithmeticError:

print("arith")

except:

print("some")

**Exercício 2**

Qual é o output esperado do seguinte código?

try:

print(1/0)

except ArithmeticError:

print("arith")

except ZeroDivisionError:

print("zero")

except:

print("some")

**Exercício 3**

Qual é o output esperado do seguinte código?

def foo(x):

assert x

return 1/x

try:

print(foo(0))

except ZeroDivisionError:

print("zero")

except:

print("some")

# Exceções integradas

Vamos mostrar-lhe uma pequena lista das exceções mais úteis. Embora possa parecer estranho chamar "útil" a uma coisa ou fenómeno que é um sinal visível de fracasso ou contratempo, como sabe, errar é humano, e se algo pode correr mal, correrá mal.

As exceções são tão rotineiras e normais como qualquer outro aspeto da vida de um programador.

Para cada exceção, mostraremos:

* o seu nome;
* a sua localização na árvore de exceção;
* uma breve descrição;
* um snippet conciso de código mostrando as circunstâncias em que a exceção pode ser levantada.

Há muitas outras exceções a explorar - simplesmente não temos espaço para as explorar todas.

## ArithmeticError

**Localização:** BaseException ← Exception ← ArithmeticError

**Descrição**: uma exceção abstrata incluindo todas as exceções causadas por operações aritméticas como a divisão zero ou o domínio inválido de um argumento

## AssertionError

**Localização:** BaseException ← Exception ← AssertionError

**Descrição**: uma exceção concreta levantada pela instrução assert quando a sua argumentação avalia para False, None, 0, ou uma string vazia

**Código:**

from math import tan, radians

angle = int(input('Enter integral angle in degrees: '))

# We must be sure that angle != 90 + k \* 180

assert angle % 180 != 90

print(tan(radians(angle)))

## BaseException

**Localização:** BaseException

**Descrição:** mais geral (abstrata) de todas as exceções Python - todas as outras exceções estão incluídas nesta; pode dizer-se que os doss seguintes except ramos são equivalentes: except: e except BaseException:.

## IndexError

**Localização:** BaseException ← Exception ← LookupError ← IndexError

**Descrição:** uma exceção concreta levantada quando se tenta aceder a um elemento de uma sequência inexistente (por exemplo, o elemento de uma lista)

**Código:**

# The code shows an extravagant way

# of leaving the loop.

the\_list = [1, 2, 3, 4, 5]

ix = 0

do\_it = True

while do\_it:

try:

print(the\_list[ix])

ix += 1

except IndexError:

do\_it = False

print('Done')

## KeyboardInterrupt

**Localização:** BaseException ← KeyboardInterrupt

**Descrição**: uma exceção concreta levantada quando o utilizador utiliza um atalho de teclado concebido para terminar a execução de um programa (*Ctrl-C* na maioria dos sistemas operativos); se o tratamento desta exceção não levar à conclusão do programa, o programa continua a sua execução.

Nota: esta exceção não é derivada da Exception classe. Execute o programa em IDLE.

**Código:**

# This code cannot be terminated

# by pressing Ctrl-C.

from time import sleep

seconds = 0

while True:

try:

print(seconds)

seconds += 1

sleep(1)

except KeyboardInterrupt:

print("Don't do that!")

## LookupError

**Localização:** BaseException ← Exception ← LookupError

**Descrição**: uma exceção abstrata incluindo todas as exceções causadas por erros resultantes de referências inválidas a diferentes coleções (listas, dicionários, tuples, etc.)

## MemoryError

**Localização:** BaseException ← Exception ← MemoryError

**Descrição:** uma exceção concreta levantada quando uma operação não pode ser concluída devido a uma falta de memória livre.

**Código:**

# This code causes the MemoryError exception.

# Warning: executing this code may affect your OS.

# Don't run it in production environments!

string = 'x'

try:

while True:

string = string + string

print(len(string))

except MemoryError:

print('This is not funny!')

## OverflowError

**Localização:** BaseException ← Exception ← ArithmeticError ← OverflowError

**Descrição**: uma exceção concreta levantada quando uma operação produz um número demasiado grande para ser armazenado com sucesso

**Código:**

# The code prints subsequent

# values of exp(k), k = 1, 2, 4, 8, 16, ...

from math import exp

ex = 1

try:

while True:

print(exp(ex))

ex \*= 2

except OverflowError:

print('The number is too big.')

## ImportError

**Localização:** BaseException ← Exception ← StandardError ← ImportError

**Descrição**: uma exceção concreta levantada quando uma operação de importação falha

**Código:**

# One of these imports will fail - which one?

try:

import math

import time

import abracadabra

except:

print('One of your imports has failed.')

## KeyError

**Localização:** baseException ← Exceção ← LookUpError ← KeyError

**Descrição**: uma exceção concreta levantada quando se tenta aceder ao elemento inexistente de uma coleção (por exemplo, o elemento de um dicionário)

**Código:**

# How to abuse the dictionary

# and how to deal with it?

dictionary = { 'a': 'b', 'b': 'c', 'c': 'd' }

ch = 'a'

try:

while True:

ch = dictionary[ch]

print(ch)

except KeyError:

print('No such key:', ch)

Por agora acabámos com as exceções, mas voltarão quando discutirmos a programação orientada ao objeto em Python. Pode utilizá-las para proteger o seu código contra acidentes graves, mas também tem de aprender a mergulhar nelas, explorando a informação que transportam.

As exceções são de facto objetos - no entanto, nada podemos dizer sobre este aspeto até lhe apresentarmos classes, objetos, e afins.

Por enquanto, se quiser saber mais sobre exceções por si próprio, consulte a Biblioteca Padrão Python em <https://docs.python.org/3.6/library/exceptions.html>.

# LAB Ler Int’s em segurança

Médio

## Objetivos

* Melhorar as competências do aluno na definição de funções;
* utilizar exceções a fim de proporcionar um ambiente de input seguro.

## Cenário

A sua tarefa é a de escrever uma **função capaz de fazer input de valores inteiros e de verificar se eles estão dentro de um intervalo especificado**.

A função deve:

* aceitar três argumentos: um prompt, um limite inferior aceitável e um limite superior aceitável;
* se o utilizador inserir uma string que não é um valor inteiro, a função deve emitir a mensagem Error: wrong input, e pedir ao utilizador para inserir o valor novamente;
* se o utilizador introduzir um número que fica fora do intervalo especificado, a função deve emitir a mensagem Error: the value is not within permitted range (min..max) e pedir ao utilizador para inserir o valor novamente;
* se o valor de input for válido, devolve-o como um resultado.

## Dados de teste

Teste o seu código com cuidado.

É assim que a função deve reagir ao input do utilizador:

Enter a number from -10 to 10: 100

Error: the value is not within permitted range (-10..10)

Enter a number from -10 to 10: asd

Error: wrong input

Enter number from -10 to 10: 1

The number is: 1

# LAB PASTES

# LER INTS EM SEGURANÇA

def read\_int(prompt, min, max):

ok = False

while not ok:

try:

value = int(input(prompt))

ok = True

except ValueError:

print("Error: wrong input")

if ok:

ok = value >= min and value <= max

if not ok:

print("Error: the value is not within permitted range (" + str(min) + ".." + str(max) + ")")

return value;

v = read\_int("Enter a number from -10 to 10: ", -10, 10)

print("The number is:", v)

**Key takeaways**

1. Algumas exceções abstratas incorporadas em Python são:

* ArithmeticError,
* BaseException,
* LookupError.

2. Algumas exceções concretamente incorporadas em Python são:

* AssertionError,
* ImportError,
* IndexError,
* KeyboardInterrupt,
* KeyError,
* MemoryError,
* OverflowError.

**Exercício 1**

Qual das exceções irá utilizar para proteger o seu código de ser interrompido pelo uso do teclado?

Verifique

**Exercício 2**

Qual é o nome da exceção mais geral de todas em Python?

Verifique

**Exercício 3**

Qual das excepções será levantada através da seguinte avaliação mal sucedida?

huge\_value = 1E250 \*\* 2

Verifique

**Parabéns! Completou o *PE2: Módulo 2*.**

Muito bem! Chegou ao fim do Módulo 2 e completou um marco importante na sua educação em programação Python. Aqui está um breve resumo dos objetivos que abordou e com os quais se familiarizou no Módulo 2:

* carateres, strings, e padrões de codificação;
* a natureza das strings em Python; strings vs. listas - semelhanças e diferenças;
* métodos de lista e string;
* manipulação de erros em Python;
* controlar o fluxo de erros usando *try* e *except*;
* a hierarquia das exceções; revisão das exceções mais úteis.

Está agora pronto para fazer o quiz do módulo e tentar o desafio final: Teste do Módulo 2, que o ajudará a avaliar o que aprendeu até agora.

