

Normalização Unicode em Python

Além da **normalização Unicode** e as formas de normalização NFC, NFD, NFKC e NFKD, você vai aprender tudo o que precisa saber sobre o padrão Unicode em si e Python.

Falando sobre a normalização Unicode em si, que provavelmente é o que te trouxe aqui: normalizar é o ato de transformar strings (**textos no padrão unicode**) para uma forma normal onde os caracteres sempre terão a mesma representação binária em todo o seu programa. Isso facilita a comparação, indexação e ordenação de strings já que, em um sistema "normalizado", essas operações são é mais confiáveis.

Frequentemente você verá emojis no meio do texto com um código na frete. Eu espero que você entenda isso ao terminar sua leitura, porque eu não costumo escrever assim, ok? 🤖 (U+1F910).

Um contexto para iniciarmos

Vamos iniciar uma jornada longa neste momento. Portanto, vou te deixar um contexto para discutirmos ao longo de todo o artigo. Porém, não se preocupe se não entender nada agora. Prometo que vou explicar tudo o que você vai ver a seguir 🙏 (U+1F64F).

Porque precisamos de normalização?

No padrão Unicode, caracteres são representados por **code points** (códigos de identidade do caractere). Mas, alguns desses caracteres são representados mais de uma vez para que o padrão Unicode mantenha compatibilidade com outros padrões que vieram antes dele.

Por exemplo, a letra "á" (a com acento agudo), representada por "U+00E1" em **code point** Unicode, também pode ser representada por "U+0061" (a) +

"U+0301" (acento agudo). Na segunda representação, o acento é algo que é chamado de **combining character**, porque combinado ao "a", forma "á". No entanto, "U+00E1" (á) não é igual a "U+0061" + "U+0301" (á) do ponto de vista do seu programa em Python, mesmo que visualmente o caractere final seja exatamente o mesmo (á).

```
>>> '\u00e1'
'á'
>>> '\u0061\u0301'
'á'
>>> '\u00e1' == '\u0061\u0301'
False
```

A **normalização unicode** vai resolver este problema mantendo apenas uma forma normal dos "ás" apresentados acima. Ou "U+00E1" ou "U+0061" + "U+0301". No entanto, para entender porque precisamos de normalização unicode em nosso sistema, precisamos entender o Padrão Unicode como um todo.

Unicode – o básico do básico

O Unicode é um padrão que permite aos computadores representar e manipular texto de qualquer sistema de escrita existente utilizando códigos para caracteres individuais. Cada caractere é mapeado para um código específico chamado de *"code point"*.

Code Points são representados por um **U+** seguido de **4 a 6 dígitos hexadecimais** (de 0 a 0x10FFFF). Por exemplo: o code point **U+0041** representa a letra "A"; o **U+0042**, a letra "B", o **U+1F40D**, uma cobra verde "🐍", e assim por diante.

Para que um sistema possa representar um **code point** como um caractere "normal", ele precisa de um sistema de **codificação de caracteres**. Este sistema de codificação também é provido pelo padrão Unicode e é responsável por representar uma sequência de **code points** (qualquer string no padrão Unicode) como um conjunto de **code units** na memória do computador, que então são mapeados para bytes de 8-bits.

Apesar do padrão Unicode disponibilizar um conjunto razoavelmente grande de sistemas de codificação de caracteres, como UTF-7, UTF-8, UTF-EBCDIC, UTF-16 e UTF32, a codificação mais usada atualmente é a **UTF-8** (UTF sendo **Unicode**

[Transformation Format](#) e 8 sendo o número de bits por código). No momento da escrita deste post, o site [W3Techs – Historical trends in the usage statistics of character encodings for websites](#), mostra o padrão UTF-8 sendo usado em **94.7%** dos sites analisados até 15/04/2020 🗓️ (U+1F440).

É uma boa ideia manter seu editor de códigos ou IDE no padrão **UTF-8** para digitar seus códigos em Python 🙌 (U+1F937).

Python e Unicode

Dica 💡 (U+1F4A1): Boa parte do trecho a seguir foi baseada na [documentação oficial do Python](#).

As strings (`str`) em Python contém caracteres Unicode desde a versão 3.0. Isso quer dizer que qualquer valor entre aspas simples, duplas ou triplas são salvas em Unicode. De fato, o Python 🐍 suporta até mesmo identificadores com caracteres Unicode.

```
>>> atenção = 'Um teste unicode'
>>> atenção
'Um teste unicode'
>>>
```

Usando caracteres unicode

Também existem várias maneiras para usar caracteres Unicode dentro do código 🖥️ (U+1F4BB).

Por exemplo, você pode usar o caractere literal (como de costume), mas também pode usar o nome do caractere Unicode, um hexadecimal ou até um número decimal que representaria o caractere usando função `chr`:

```
>>> 'A' # Caractere literal A
'A'
>>> chr(0x41) # Usando a função chr com hexadecimal
'A'
>>> chr(65) # Usando a função chr com decimal
'A'
>>> '\N{Latin Capital Letter A}' # Usando o nome do caractere
'A'
```

```
>>> '\u0041' # Usando um hexadecimal 16-bit
'A'
>>> '\U00000041' # Usando um hexadecimal 32-bit
'A'
```

Veja acima, que representei a letra “A” de várias maneiras diferentes.

Obtendo valores Unicode dos caracteres

Além do que descrevi anteriormente, você também pode fazer o inverso, ou seja, pegar os valores decimal e hexadecimal que representam o caractere desejado.

Para isso, você pode usar as funções `ord` e `hex`, dependendo do que deseja (talvez seja necessário combiná-las).

Por exemplo:

```
>>> ord('A') # Obtém o valor decimal que representa A
65
>>> hex(ord('A')) # Obtém o valor hexadecimal que representa A
'0x41'
```

Encode (str) e Decode (bytes)

É possível converter uma string em bytes ou bytes em string usando os métodos `encode` da `string` ou `decode` de `bytes`. Esses métodos recebem dois argumentos. O primeiro argumento especifica a codificação de caracteres desejada (`utf-8` ou qualquer outra disponível em [Standard Encodings](#) – use `utf-8` sempre que possível 🧐). O segundo informa como os erros devem ser tratados (falaremos desse argumento mais adiante neste post).

Porém, é importante tomar cuidado ao converter uma codificação de caracteres para outra (exemplo, de **ASCII** para **UTF-8**). Pode não ser possível mapear o código de um caractere para outro em determinadas circunstâncias.

Veja exemplos a seguir.

Encode (str)

Suponha que eu queira converter uma string **UTF-8** para bytes **UTF-8**.

Eu posso fazer isso da seguinte maneira:

```
>>> meu_nome = 'Otávio'
>>> meu_nome_em_bytes = meu_nome.encode('utf-8')
>>> meu_nome_em_bytes
b'Ot\xc3\xalvio'
>>>
```

Bytes são representados por `b'valores'` em Python.

De acordo com o código anterior, tudo ocorreu perfeitamente. Isso porque converti uma string sabendo que ela tinha caracteres **UTF-8** para bytes em **UTF-8**. Mantendo a mesma codificação de caractere, não terei problemas.

Mas, eu também poderia querer converter minha string **UTF-8** para **ASCII** (um outro tipo de codificação de caracteres). Dependendo do que você estiver convertendo, não terá problemas, porque o UTF-8 foi feito para ser compatível com outras codificações de caracteres existentes. Porém, assim que um caractere sair do range suportado pelo ASCII (de 0 a 127 em base 10), terei um erro:

```
>>> meu_nome_em_bytes = meu_nome.encode('ascii')
Traceback (most recent call last):
...
UnicodeEncodeError: 'ascii' codec can't encode character '\xe1' in ;
```

Veja que no erro é descrito o problema. Não foi possível codificar o caractere `'\xe1'` na posição 2.

Lembra que te mostrei como exibir o caractere utilizando a função `chr`? Então, o caractere `\xe1` é o mesmo que `chr(0xe1)`, ou `"á"` de **"Otávio"**. Esse caractere não faz parte da tabela **ascii**, portanto o erro.

Logo mais veremos o segundo argumento e você poderá selecionar o que acontece quando um erro assim ocorrer.

Decode (bytes)

Se o método `encode` é usado para converter string em bytes, o método `decode` é usado para fazer o inverso disso, converter bytes em strings.

Por exemplo, suponha que eu tenha apenas os bytes e queira resgatar seu valor para uma string **UTF-8**.

```
>>> meu_nome_em_bytes = b'Ot\xc3\xalvio'
>>> meu_nome_str = meu_nome_em_bytes.decode('utf-8')
>>> meu_nome_str
'Otávio'
```

Novamente, aqui ocorreu tudo perfeitamente, porque eu sabia que a codificação dos bytes era **UTF-8** e eu os decodifiquei adequadamente. Então tudo ocorreu como esperado. Mas, e se eu quisesse decodificar esses bytes em **ASCII**?

Inicialmente, não tem como (😞 – U+1F612)!

Para isso você precisa saber a codificação de caracteres usada na codificação para decodificar.

Dica: `chardet` ajuda a tentar descobrir a codificação de caracteres usada em bytes que você não saberia de outra forma. Mas a maneira mais simples continua sendo: “pergunte ao dono dos bytes qual a codificação”. **UTF-8** é um bom chute inicial.

Erros de codificação em decode (bytes)

Imagine que eu não saiba a codificação usada em determinados bytes e tente chutar **ascii**, por exemplo:

```
>>> meu_nome_str = meu_nome_em_bytes.decode('ascii') # Otávio (em U
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in positio
```

Perceba que aqui, além de um erro de `UnicodeDecodeError`, eu também tenho um **code point** incorreto. Se você observar, `0xc3` aponta para “**Ã**”, que nem existia na minha string anterior.

O motivo disso é simples, “**á**” da minha string anterior usa dois bytes em **UTF-8** e o codec **ASCII** não sabe disso. Então ele tenta decodificar byte por byte e gera

esse erro estranho. Se isso passasse sem erros, o resultado seria um monte de caracteres que não fariam sentido algum. Por exemplo, se eu usasse o codec `latin1` ao invés de `ascii`, o resultado seria `'OtÃ¡vio'`.

Para contornar essa situação, eu preciso saber qual a codificação de caracteres foi usada para codificar os bytes anteriormente. Sabendo disso, eu deveria decodificar esses bytes usando a codificação correta antes de fazer qualquer outra coisa.

Depois de decodificar, eu poderia codificar novamente usando o **codec** que eu preferir, contando que ele suporte os caracteres que eu estiver usando (usa sempre **UTF-8**, pelo amor de Deus 🙏 – U+1F62C).

Por exemplo, se eu quero codificar de UTF-8 para `ISO-8859-1 (Latin1)`, que é algo que vejo muito aqui no Brasil, principalmente em sistemas públicos, poderia fazer assim:

```
>>> meu_nome_str = meu_nome_em_bytes.decode('utf8') # Otávio
>>> meu_nome_em_bytes_latin = meu_nome_str.encode('latin_1') # Otáv
>>> meu_nome_em_bytes_latin
b'Ot\xelvio' # Otávio
```

Basicamente, isso foi uma conversão de **UTF-8** para `latin_1`. Tenha noção de que sempre que essas conversões ocorrem, uma codificação de caracteres deve suportar a outra. O Unicode foi criado para ser compatível com todas as codificações existentes. No entanto, apenas no sentido de “qualquer codificação” convertido para “Unicode”. Você pode ter problemas ao converter no sentido contrário, de Unicode para “qualquer codificação”, porque o padrão Unicode suporta muito mais caracteres do que qualquer outra codificação de caracteres que você quiser utilizar.

No nosso exemplo, tudo funcionou perfeitamente porque todas as letras de `“Otávio”` estão presentes na tabela `ISO-8859-1 (Latin1)`, caso contrário ocorreriam erros também.

Dicas

Dica número 1: Sempre que possível use a codificação de caracteres **UTF-8**, na grande maioria das vezes isso é possível 🙏 (U+1F605).

Mais dicas: se você precisa detectar a codificação de caracteres de algo que não tem a mínima ideia como foi codificado, use [charset.detect](#). Ele não vai acertar em 100% dos casos, mas já me salvou de muitas enrascadas; Se você precisa saber quais codecs de codificação o Python suporta, veja [Python Specific Encodings](#).

Erros em encode e decode

Como te contei anteriormente, `encode` e `decode` recebem um argumento com a codificação desejada e outro especificando como os erros devem ser tratados. Para o segundo argumento você pode enviar os seguintes valores:

- `'strict'` – É o padrão. O que levanta uma exceção de [UnicodeEncodeError](#) ou [UnicodeDecodeError](#);
- `'replace'` – Usa o caractere U+FFFD (REPLACEMENT CHARACTER) no lugar do caractere que não pôde ser convertido;
- `'ignore'` – Simplesmente pula o caractere que não pode ser convertido;
- `'backslashreplace'` – que insere uma sequência `\xNN` no lugar do caractere que não pode ser convertido;
- `'xmlcharrefreplace'` – que insere uma referência para um caractere [XML](#) (isso só funciona com `encode`).

```
>>> 'Otávio'.encode('utf-8')
b'Ot\xc3\xalvio'
>>> b'Ot\xc3\xalvio'.decode('ascii', 'ignore') # aqui usei ignore
'Otvio' # e perdi o "á"
```

Normalização Unicode em Python

Nós demos várias voltas até chegar aqui, mas é importante conhecer o que você está fazendo, não é mesmo (🤔 – U+1F60F)?

Então, só pra recapitular tudo:

- Você conheceu os code points do Unicode;
- Também sabe que Unicode foi feito pensando em compatibilidade com padrões já existente (ascii, latin, etc). Vamos voltar nesse assunto já já;
- Viu que UTF-8 é uma das codificações de caracteres do Unicode;

- Está ciente que UTF-8 é, de longe, uma das codificações mais usadas no mundo;
- E deveria estar usando UTF-8 nos seus código (é muito provável que já esteja).

Uma coisa que eu ainda não te falei é sobre a normalização e o porquê isso existe.

Na verdade, todas as voltas foram para fazer você entender o que é Unicode de verdade. Se já sabia, melhor ainda!

Então agora podemos ter uma conversa mais “complexa”.

Unicode e outros padrões

Lembra que lá no comecinho 🙌 (U+261D) te falei que eu poderia escrever a letra “á” de maneiras diferentes em Unicode?

Só pra te lembrar:

```
>>> '\u00e1'
'á'
>>> '\u0061\u0301'
'á'
```

Este pode não ser um problema no seu programa caso não tenha tido a necessidade de comparar esses dois “ás”. No entanto, em algum momento este problema pode aparecer e você vai demorar um tempo considerável até descobrir que isso está relacionado com a falta de normalização de caracteres. Nós buscamos recursos de várias fontes externas ao nosso programa e não sabemos qual forma normal utilizaram no sistema deles, e essa bomba pode explodir na sua mão.

O Unicode foi criado pensando em compatibilidade, por isso alguns caracteres aparecem mais de uma vez. Por exemplo, se você olhar na tabela [ASCII](#), vai ver que a letra “A” é representada pelo mesmo hexadecimal que o Unicode (**41** vs **U+0041**). Se olhar na tabela [ISO/IEC 8859-1](#), vai ver que a letra “á” é representada exatamente pelo mesmo hexadecimal que o Unicode (**00E1** vs **U+00E1**). Isso quer dizer que o range de 0 a 127 (base 10) no Unicode é compatível com **ASCII**, o range de 0 a 255 (base 10) no Unicode é compatível com **ISO/IEC 8859-1** (ou

latin1) e assim por diante. O Unicode tenta ser compatível com todos os padrões existentes.

Isso vai acabar nos levando a um problema, vai vendo 🤔 (U+1F928)!

Caracteres pré-compostos e caracteres combinados

Pelo motivo que te expliquei anteriormente, existem caracteres que são chamados de **pré-compostos**, como: **á, é, À, Á** e vários outros. Esses caracteres pré-compostos existem para manter compatibilidade com padrões que já existiam antes do Unicode.

Por outro lado, o Unicode também dispõe de um sistema de combinação para estender o repositório de caracteres suportados, e isso é genial (🥰 – U+1F970)!

Pensa comigo 🤪 (U+1F913), se eu posso ter um “a” e um “**acento agudo**” em dois caracteres diferentes, não seria inteligente permitir que o **acento agudo** pudesse ser combinado com esse “a” ou com qualquer outro caractere formando um caractere único? Também acho!

É exatamente esse o mecanismo que foi usado no Unicode. Ao invés de ter um **code point** único para cada caractere do planeta, fizeram um sistema de combinação de caracteres para formar esses símbolos loucos que a gente acaba usando e nem percebe.

Esses caracteres que podem ser combinados com outros caracteres são chamados de **combining character** e existem muitos deles.

Mas, como nem tudo são flores (🌺 – U+1F940), isso gerou o problema de ter mais de um caractere representando a mesma coisa. Aquela probleminha que te mostrei no início, sobre os “ás”. Te falei que ia dar problema, não falei 😊 (U+1F601)?

É aqui que entra a normalização e uma outra coisa que é chamada de **equivalência canônica**.

Equivalência canônica

Como os criadores do Unicode são bem inteligentes 🤖 (U+1F9D0), eles criaram algo chamado de “**equivalência canônica**”. Isso é só uma maneira bonita de falar “esses dois caracteres são iguais”. Então, na equivalência canônica, **U+00E1** (á pré-

composto) é igual a **U+0041 + U+0301** (a com **acento agudo** combinados). Isso acontece com todos os caracteres acentuados e mais outros milhares de caracteres que podem ser combinados em vários idiomas diferentes.

Sabendo disso, você poder utilizar mais de uma forma normal em todo o seu programa: **NFC** e **NFD** (tem mais duas, mas é questão de compatibilidade, segura aí que a gente já fala sobre isso).

NFC – Normalization Form Canonical Composition

Esse tipo de normalização Unicode visa **manter os caracteres pré-compostos** no seu programa (sem a separação de caractere + caractere combinado). Tais caracteres são unidos por equivalência canônica.

Lembra dos **ás**? Aqui eles serão iguais, porque somente o **U+00E1** (**á** pré-composto) será mantido, os caracteres separados serão convertidos em pré-compostos. Por exemplo, **U+0061 + U+0301** (a com **acento agudo** combinados) se tornaria sempre **U+00E1** (**á** pré-composto).

Por exemplo:

```
>>> import unicodedata
>>> nome = 'Ot\u0061\u0301vio'
>>> nome_normalizado = unicodedata.normalize('NFC', nome)
>>> ['U+' + hex(ord(letra))[2:].zfill(4).upper() for letra in nome_
['U+004F', 'U+0074', 'U+00E1', 'U+0076', 'U+0069', 'U+006F']
# O          t          á          v          i          o
```

Da pra perceber ali que a letra “**á**” do meu nome, sempre será mantida como **U+00E1** com esse tipo de normalização. Mesmo eu dizendo explicitamente que quero a string `'Ot\u0061\u0301vio'`.

Resumidamente: isso não fará nada com caracteres pré-compostos, mas combinará caracteres equivalentes que estiverem separados em sua forma pré-composta por equivalência canônica.

NFD – Normalization Form Canonical Decomposition

Esse tipo de normalização unicode visa manter os caracteres separados (com a separação entre caractere e caractere combinado). Os caracteres serão separados

por equivalência canônica.

Aqui os “**ás**” serão iguais, porque somente os caracteres **U+0061 + U+0301** (a com **acento agudo** combinados) serão mantidos. Os “**ás**” pré-compostos (**U+00E1**) serão convertidos em **U+0061 + U+0301**.

Por exemplo:

```
>>> import unicodedata
>>> nome = 'Ot\u00e1lvio'
>>> nome_normalizado = unicodedata.normalize('NFD', nome)
>>> ['U+' + hex(ord(letra))[2:].zfill(4).upper() for letra in nome_
['U+004F', 'U+0074', 'U+0061', 'U+0301', 'U+0076', 'U+0069', 'U+006
# O          t          a          acento        v          i          o
```

Perceba que agora eu consegui manter ambos os caracteres, tanto o “**a**” quanto o “**acento agudo combinado**”. Mesmo especificando que eu queria a string 'Ot\u00e1lvio'.

Resumidamente: isso não fará nada com aqueles dois caracteres combinados, porém vai separar caracteres pré-compostos para sua forma combinada por equivalência canônica. Basicamente é **U+00E1** (á pré-composto) se transformando em **U+0061 + U+0301** (a com **acento agudo** combinados).

NFKC e NFKD – Normalization Form Compatibility Composition/Decomposition

Para complicar um pouquinho mais a sua vida na normalização unicode, também existem caracteres que **não** são definidos por **equivalência canônica**, mas por **compatibilidade**.

Por exemplo, em alguns contextos, o símbolo **TM** pode ter o mesmo significado que TM (TRADE MARK SIGN, U+2122). Nesse caso, ambos TM e TM são definidos como caracteres compatíveis, mas que **NÃO** TEM **equivalência canônica**.

Isso quer dizer que nem **NFC**, nem **NFD** vão normalizar esses dois valores.

E só pra deixar claro isso pra você, caso ainda não tenha ficado:

- NF – Normalization Form (formato de normalização);

- C – Composition (composição – une);
- D – Decomposition (decomposição – separa);
- K – Compatibility (separa por compatibilidade).

Agora que vem a pergunta de 1 milhão de dólares: qual a forma normal entre TM e ™? Depende! Em qual contexto?

Vou te dar um exemplo: nós sabemos que seres humanos tem uma preguiça danada de digitar as coisas corretamente, certo? Imagine que a minha marca fosse **OM™** e eu quisesse que no meu sistema de busca, essa marca fosse encontrada. Você acha que as pessoas digitariam **OMTM** ou **OM™**? Eu acho que OMTM (caso não encontrassem antes apenas digitando OM). Mas podemos garantir as duas com a normalização.

Então nesse caso, eu usaria a compatibilidade para transformar ™ em TM apenas para realizar uma comparação. Por exemplo, meu texto na base de dados seria normalizado temporariamente com NFKD e o texto enviado pelo usuário também seria normalizado para NFKD. Assim eu consigo encontrar **OMTM** ou **OM™** independente de como isso foi digitado pelo usuário.

Para fazer a normalização unicode de ™ para TM, você vai precisar usar NFK(C ou D):

```
>>> import unicodedata
>>> nome = 'OM™'
>>> nome_normalizado = unicodedata.normalize('NFKC', nome)
>>> ['U+' + hex(ord(letra))[2:].zfill(4).upper() for letra in nome_
['U+004F', 'U+004D', 'U+0054', 'U+004D']
# O           M           T           M
```

Perceba que “**C**” e “**D**” aqui vão fazer o mesmo trabalho descrito anteriormente, mas o “**K**” vai trabalhar na compatibilidade que te falei antes.

Então só pra resumir: O **K** significa **compatibility** e vai converter valores que estariam em apenas um **code point** Unicode em caracteres que seriam compatíveis (de acordo com as regras deles, que eu não sei quais são). Esses caracteres devem se comportar da mesma maneira em pesquisa, comparação, ordenação e indexação, mas podem mudar o significado e também podem parecer visualmente diferentes em vários contextos. Como no nosso exemplo,

OMTM ou **OM™** deveria retornar os mesmos valores no meu sistema de pesquisa ou comparação, mas eles são bem diferentes visualmente.

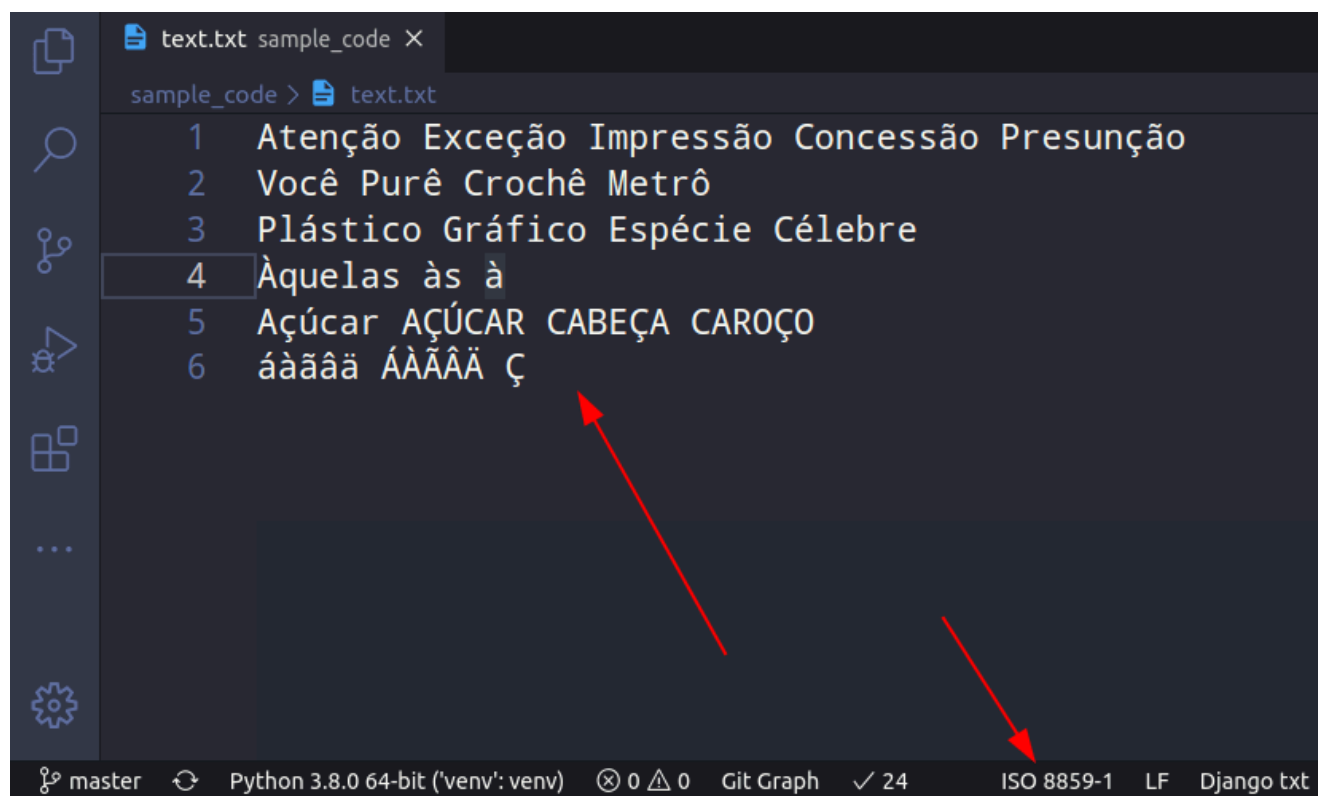
Um ponto de atenção para K

Tenha em mente que a partir do momento que eu separei os valores por compatibilidade, não consigo mais uni-los novamente. Por exemplo, se eu normalizar com **K** (NFKC ou NFKD) o valor **™** (TRADE MARK SIGN, U+2122), vou obter **TM** (como vimos). Porém **TM** não voltará a ser **™**.

Por este motivo, é super importante que você **não** salve os valores permanentemente utilizando **K**. Você deve normalizar temporariamente no momento que precisar realizar alguma comparação e eliminar esse valor após terminar o que estava fazendo. Salve os valores como eles realmente são.

Usando chardet para detectar a codificação de caracteres

Na grande maioria das vezes que nosso sistema gerar algum problema de codificação de caracteres, esse problema virá de algum recurso externo. Portanto, para simular isso, suponha que eu tenha um arquivo em **ISO-8859-1** (latin1). Qualquer editor de textos decente vai te permitir criar o mesmo texto com a mesma codificação de caracteres. Por exemplo, o [Visual Studio Code](#).



```
text.txt sample_code X
sample_code > text.txt
1  Atenção Exceção Impressão Concessão Presunção
2  Você Purê Crochê Metrô
3  Plástico Gráfico Espécie Célebre
4  Àquelas às à
5  Açúcar AÇÚCAR CABEÇA CAROÇO
6  áääää ÁÄÄÄÄ Ç
```

master Python 3.8.0 64-bit ('venv': venv) 0 0 Git Graph ✓ 24 ISO 8859-1 LF Django txt

Nós sabemos qual a codificação de caracteres foi usada neste arquivo (latin1), mas finja que não. Vamos carregar esse arquivo pelo Python usando "rb" (read bytes) e solicitar ao **chardet** para detectar a codificação de caracteres.

Nota: você precisa instalar o chardet com "pip install chardet".

```
>>> import chardet
>>> with open('text.txt', 'rb') as file:
...     raw_content = file.read()
...     encoding = chardet.detect(raw_content)
...     encoding["encoding"]
...
'ISO-8859-9'
```

Ele detectou como 'ISO-8859-9', isso seria **latin5** e não **latin1**, mas lembra que te falei que ele não iria acertar 100% das vezes, certo? Bom, vamos tentar converter esse arquivo de ISO-8859-9 (mesmo não sendo a codificação exata do arquivo) para UTF-8 e ver o que ocorre.

Vamos abrir o arquivo novamente, decodificar com a codificação que o **chardet** quiser (no caso ISO-8859-9, latin5), depois vamos abrir um novo arquivo com 'wb' e salvar como UTF-8. Veja:

```
>>> with open('text.txt', 'rb') as file:
...     # Vamos ler apenas bytes do arquivo
...     raw_content = file.read()
...     # Agora a gente decodifica
...     content_string = raw_content.decode(encoding["encoding"])
>>>
# Perfeito, agora vamos tentar pegar o conteúdo
# da content_string e salvar em outro arquivo
# porém, agora vamos codificar em UTF-8
>>> with open('text2.txt', 'wb') as file:
...     file.write(content_string.encode('utf8'))
...
192
>>>
```

Perfeito, sem erros! Agora vamos ver como está o nosso arquivo "**text2.txt**" (o novo arquivo gerado). Será que os caracteres se mantiveram?

```
sample_code > text2.txt
1  Atenção Exceção Impressão Concessão Presunção
2  Você Purê Crochê Metrô
3  Plástico Gráfico Espécie Célebre
4  Àquelas às à
5  Açúcar AÇÚCAR CABEÇA CAROÇO
6  áääää ÄÄÄÄ Ç
```

er Python 3.8.0 64-bit ('venv': venv) 1 0 Git Graph ✓ 24 UTF-8 LF Django txt Go Live 1.7 hrs

Perfeito! Viu como a aproximação que o **chardet** encontrou me ajudou muito? Mesmo que ele não tenha detectado com 100% de certeza qual a codificação de caracteres usada no arquivo, ele me passou uma que provavelmente iria funcionar.

Se você fosse tentar decodificar direto com UTF-8, isso ocorreria:

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe7 in
position 4: invalid continuation byte
```

E se ignorasse os erros, seu texto ficaria assim:

```
Ateno Exceo Impresso Concesso Presuno
Voc Pur Croch Metr
Plstico Grfico Espcie Clebre
quelas s
Acar ACAR CABEA CAROO'
```

Eu acho que deu pra você entender, não é?

Funções interessantes com normalização unicode

Você, como programador(a), já pode ter imaginado milhares de coisas interessantes que pode fazer com a normalização unicode, não é mesmo? Se não, vou te dar algumas ideias:

Obtendo code points Unicode

Suponha que eu queira obter uma lista com todos os code points de uma frase. Veja que legal:

```
from typing import List
from unicodedata import normalize

def get_unicode_code_points(string: str) -> List[str]:
    string_normalized = normalize('NFD', string)
    code_points: List[str] = [
        'U+' + hex(ord(letter))[2:].zfill(4).upper()
        for letter in string_normalized
    ]
    return code_points

if __name__ == "__main__":
    text = 'Python 🐍™'
    code_points = get_unicode_code_points(text)
    print(code_points)

    """
    ['U+0050', 'U+0079', 'U+0074', 'U+0068',
     'U+006F', 'U+006E', 'U+0020', 'U+1F40D',
     'U+2122']
    """
```

Obtendo caracteres de code points

Mas, e o inverso? Dado um code point, como converto em caractere? Você já viu isso ao longo do texto todo, mas aqui vai.

```
from typing import List

def get_char_from_code_point(code_points: List[str]) -> str:
    chars = [chr(int(c.replace('U+', '0x'), 16)) for c in code_poin
    return ''.join(chars)

if __name__ == "__main__":
```

```
code_points = [
    'U+0050', 'U+0079', 'U+0074', 'U+0068',
    'U+006F', 'U+006E', 'U+0020', 'U+1F40D',
    'U+2122'
]

print(get_char_from_code_point(code_points))
# Python 🐍™
```

Removendo caracteres fora da tabela ASCII

Em alguns casos, pode ser interessante manter apenas caracteres compatíveis com a tabela "ASCII". Além disso, nós também podemos converter caracteres que seriam compatíveis se não fossem pré-compostos. Por exemplo, 'á', 'ã', à e â se tornariam simplesmente 'a' e assim por diante para todos os caracteres. Porém, caracteres como 🐍 e 😊 não estariam presentes, porque não existem na tabela ASCII e também não existem compatíveis.

Vamos ver como faríamos isso:

```
import unicodedata

def non_ascii_to_ascii(string: str) -> str:
    ascii_only = unicodedata.normalize('NFKD', string) \
        .encode('ascii', 'ignore') \
        .decode('ascii')
    return ascii_only

if __name__ == "__main__":
    string = 'Atenção 🐍 😊'
    print(non_ascii_to_ascii(string))  # Atencao
```

Perceba que caracteres como "ç" e "ã" de "Atenção" foram mantidos sem acento (porque existem na tabela ASCII), porém 🐍 e 😊 foram ignorados.

Removendo acentos das palavras

Na função anterior, a gente meio que removeu os acentos, porém também removemos outras coisas que não queríamos. Mas, suponha que eu queira remover apenas o **combining character** mantendo o restante (o combining character seria o acento propriamente dito).

Isso me retornaria palavras sem acento.

Eu posso fazer isso, como o [Luciano Ramalho](#) descreve em seu livro [Python Fluente](#).

```
import unicodedata

def remove_accents(string: str) -> str:
    normalized = unicodedata.normalize('NFKD', string)
    return ''.join([c for c in normalized if not unicodedata.combin
```

Porém, eu também posso fazer isso com expressões regulares. O que funcionar melhor pra você:

```
import unicodedata
import re

def remove_accents_regex(string: str) -> str:
    regex = re.compile(r'[\u0300-\u036F]', flags=re.DOTALL)
    normalized = unicodedata.normalize('NFKD', string)
    return regex.sub('', normalized)

if __name__ == "__main__":
    string = 'Atenção 🇧🇷 😊'
    print(remove_accents_regex(string))  # Atencao 🇧🇷 😊
```

Por falar nisso, eu tenho um detalhe sobre expressões regulares pra te informar: eu tenho um curso inteiro e gratuito na [Udemy](#) e no [Youtube](#) sobre isso.

Portanto, não há motivos para entrarmos em detalhes sobre [regex](#) aqui.

Mais sobre Unicode e Normalização Unicode

Eu sei que esse assunto talvez passe despercebido para vários desenvolvedores e desenvolvedoras mundo a fora e não é culpa deles (ou nossa, também passei por isso). Em nosso meio, a maioria dos cursos, faculdades e livros que você lê para aprender a programar, infelizmente não tratam desse assunto, ou, se tratam, é de maneira superficial. Porém, como você pôde ver, eu fiz questão de deixar todos os links de onde removi todas as informações que escrevi aqui. Assim, é extremamente necessário que você leia esses links também.

Além disso, ainda faltaram algumas coisas que não consegui falar neste post. Por exemplo, casefold e tratamento de arquivos, foram coisas que não mencionei aqui, mas que são mencionadas no [Unicode HOWTO oficial do Python](#). Então, dá um jeito de ler esse artigo também.

Então é isso, te deixo aqui com um pouco mais de serviço pela frente.

Te espero no próximo post.

© Copyright - Otávio Miranda