Text vs Bytes

The concept of string is simple enough: a string is just a sequence of characters. The problem lies in the definition of character.

The best definition that we have until now doe a character is the Unicode character. The identity of a Unicode character it’s called code point . The code point is a number between a 1,114,111 (base 10). This is represented as a hexadecimal number between four and six digits and a U+. There are some examples of it: {The Letter A is represented as U+0041, Euro sign is U+20AC}.

An important thing to take into account is that the actual bytes that represent the sequence of strings depends of the encoding that is being used. This is because the encodings differ in two important aspects when it comes to represent bytes as characters: How many bytes represent it, and which code point. This is because the encoding number for the letter A in ASCII might not be the same as in Unicode, and since Unicode is larger, it needs more code points to represent all of its values.

Here is an example of encoding and decoding:

Output: X

>>>

>>> s = 'café'

>>> len(s)

4

>>> b = s.encode('utf8')

>>> b

b'caf\xc3\xa9'

>>> len(b)

5>>> b.decode('utf8')

'café'>>>

Here we can see that the representation of 'café' in Unicode takes actually less space to store in Unicode than in UTF8.

Byte Essentials:

First thing to note when it comes to bytes is that there are two main types of bytes in python, the immutable ones known as the byte type and the mutable one known as the bytearray type. No matter from which byte type we are talking about, each byte is actually an integer, a number between 0 and 255. However, the slicing of a binary sequence always produces a binary sequence of the same type-even with slices of length 1:

Five-byte sequence as bytes and as bytearray:

Output: X

>>>

>>> cafe = bytes('café', encoding='utf\_8')

>>> cafe

b'caf\xc3\xa9'

>>> cafe[0]

99

>>> cafe[:1]

b'c'

>>> cafe\_arr = bytearray(cafe)

>>> cafe\_arr bytearray(b'caf\xc3\xa9')

>>> cafe\_arr[-1:]

bytearray(b'\xa9')

>>>

Here we ca see that the byte and bytearray type are sequences of numbers between 0 and 255, the only deference between them, is that one is mutable and the other one it isn’t. Also, we can see how the same sequence of numbers can be interpreted differently depending on the type of encoding-decoding we use.

The reason why the number is between zero and 255, is because with one byte, we can count only 256 number, which is what we have if we include the 0 as the value number 256. So this makes sense then.

ASCII and range(256):

range(256):

We have seen till now that the real identity of a byte type, or a bytearray type is actually byte . That was not helpful so let’s picture it form the beginning:

A byte is the union of 8 bits. A bit is value of zero or one which is ultimately referred as the machine language. A bit can have only one value, is either a ‘1’ or a ‘0’. Seeing this, we can therefore imagine, that the more bits we have, the more combinations and the more values we can have. The standard in the world of informatic is to take bits and group them into groups of eight. This group of eight bits is balled a byte.

By combinatory, if we have two possible values for a bit, just one bitt can take possible values as we saw before. But then what happens with 8 bits?

The answer is that the number of combinations that we can achieve with 8 bits, is exactly 256, wo if we were to consider the zero, which we always do, we end up counting till 255, and the zero.

ASCII:

ASCII was for a while the way that we had of representing all the values that we needed, and it worked for a while. We realized that the number of combinations for a byte was 256 and then we used every combination, meaning every number from 0-255 to represent a character. By doing this we are limited to only this characters(from space{0} to ‘~’{255}):

!"#$%&\'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^\_`abcdefghijklmnopqrstuvwxyz{|}~

Although we had our characters, now we needed a way to separate them, yes we have the space but we also need tabs, newlines, and carriage return. For this kind of things, we use the backslash especial character. Yes, this is our first special character, and it means that the character that comes after it has a special treatment and a special meaning.

\t ----Tabulation

\n ---New Line

\r ----Carriage Return

\\ ----Actual usage of the Backslash keyword

\x ---Hexadecimal values

Hexadecimal Values:

As we saw at the beginning of this document, we describe the hexadecimal numbers as \x00, for example. We do so because we receive help from the backslash special keyword to tell the interpreter that the characters that comes next must be treated differently. In this case the whole expression x00 must be treated differently. This seems a little off but is completely necessary when it comes to print values that are not in the ASCII range. For example that’s why in the example presented before, you see b'caf\xc3\xa9': the first three bytes b'caf' are in the printable ASCII range, the last two are not.

Both bytes and bytearray support every str method except those that do formatting (format, format\_map) and a few others that depend on Unicode data, including case fold, isdecimal, isidentifier, isnumeric, isprintable, and encode. This means that you can use familiar string methods like endswith, replace, strip, translate, upper, and dozens of others with binary sequences—only using bytes and not str arguments. In addition, the regular expression functions in the re module also work on binary sequences, if the regex is compiled from a binary sequence instead of a str.

Binary sequences have a class method that str doesn’t have, called fromhex, which builds a binary sequence by parsing pairs of hex digits optionally separated by spaces:

>>> bytes.fromhex('31 4B CE A9')

b'1K\xce\xa9'

>>>

How to build the byte and bytearray types:

First of all lets remember that the core of a byte of a bytearray is that they are, well bytes, and the most simple way of interpretate them, is as numbers. That’s why the way of building a byte or a bytearray is with any sequence or an array.array for example or also, we could do it with a string and an encoding keyword argument. This are all the possible ways we can do it:

• A str and an encoding keyword argument.

• An iterable providing items with values from 0 to 255.

• A single integer, to create a binary sequence of that size initialized with null bytes.

• An object that implements the buffer protocol (e.g., bytes, bytearray, memory

view, array.array); this copies the bytes from the source object to the newly created binary sequence.

Building a binary sequence from a buffer-like object is a low-level operation that may involve type casting

Initializing bytes from the raw data of an array( low-level operation ):

Output: X

>>> import array

>>> numbers = array.array('h', [-2, -1, 0, 1, 2])

>>> octets = bytes(numbers)

>>> octets

b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00'

>>>

Creating a bytes or bytearray object from any buffer-like source will always copy the bytes. In contrast, memoryview objects let you share memory between binary data structures. To extract structured information from binary sequences, the struct module is invaluable.

Structs and Memory views:

The struct module provides functions to parse packed bytes into a tuple of fields of different types and to perform the opposite conversion, from a tuple into packed bytes. struct is used with bytes, bytearray, and memoryview objects.

The memoryview class does not let you create or store byte sequences, but provides shared memory access to slices of data from other binary sequences, packed arrays, and buffers such as Python Imaging Library (PIL) images, without copying the bytes.

Using memoryview and struct to inspect a GIF image header:

Output: X

>>>

>>> import struct

>>> fmt = '<3s3sHH' #

>>> with open('filter.gif', 'rb') as fp:

... img = memoryview(fp.read()) #

...

>>> header = img[:10] #

>>> bytes(header) #

b'GIF89a+\x02\xe6\x00'

>>> struct.unpack(fmt, header) #

(b'GIF', b'89a', 555, 230)

>>>

In this example what we are doing is extracting from the bytes of this file, the header. The file needs to be unpacked by using a format ‘<3s3sHH’ that comes directly from the structs module. Then we create a memoryview that holds into memory the bytes that we are requiring from this file. In this case, taking the bytes of img[:10] will result into getting the header of this file, which is what we are looking for to demonstrate how useful can be to take a buffer saved in memory and parse the right bytes, in the right way in order to get a specific information.

Note that slicing a memoryview returns a new memoryview, without copying bytes—one of the technical reviewers—pointed out that even less byte copying would happen if I used the mmap module to open the image as a memory-mapped file.

Basic Encoders/Decoders

The Python distribution bundles more than 100 codecs (encoder/decoder) for text to byte conversion and vice versa. Each codec has a name, like 'utf\_8', and often aliases, such as 'utf8', 'utf-8', and 'U8', which you can use as the encoding argument in functions like open(), str.encode(), bytes.decode(), and so on.

The string “El Niño” encoded with three codecs:

Output: X

>>> for codec in ['latin\_1', 'utf\_8', 'utf\_16']:

... print(codec, 'El Niño'.encode(codec), sep='\t')

...

latin\_1 b'El Ni\xf1o'

utf\_8 b'El Ni\xc3\xb1o'

utf\_16 b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'

>>>

Twelve characters, their code points, and their byte representation (in hex)

Graphical user interface, table

Description automatically generated

All those asterisks make clear that some encodings, like ASCII and even the multibyte GB2312, cannot represent every Unicode character. The UTF encodings, however, are designed to handle every Unicode code point.

Understanding Encode/Decode Problems

Although there is a generic UnicodeError exception, the error reported is almost always more specific: either a UnicodeEncodeError (when converting str to binary sequences) or a UnicodeDecodeError (when reading binary sequences into str). Loading Python modules may also generate a SyntaxError when the source encoding is unexpected. The first thing to note when you get a Unicode error is the exact type of the exception. Is it a UnicodeEncodeError, a UnicodeDecodeError, or some other error (e.g., SyntaxError) that mentions an encoding problem?

Coping with UnicodeEncodeError:

Most non-UTF codecs handle only a small subset of the Unicode characters. When converting text to bytes, if a character is not defined in the target encoding, UnicodeEncodeError will be raised, unless special handling is provided by passing an errors argument to the encoding method or function.

Here we can see how the different encoders handle this exception, Some of them just use the wrong value, and others raise an error. The problem is that we must’ve told the program how to handle the error, if we wanted to omit the error —which is not a good idea– or replace it. Replace it will return a default string which is a question mark that differs from encoder to encoder, and from one character type to the other.

Encoding to bytes: success and error handling:

Output: X

>>>

>>> city = 'São Paulo'

>>> city.encode('utf\_8')

b'S\xc3\xa3o Paulo'

>>> city.encode('utf\_16')

b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'

>>> city.encode('iso8859\_1')

b'S\xe3o Paulo'

>>> city.encode('cp437')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/.../lib/python3.4/encodings/cp437.py", line 12, in encode

return codecs.charmap\_encode(input,errors,encoding\_map)

UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in

position 1: character maps to <undefined>

>>> city.encode('cp437', errors='ignore')

b'So Paulo'

>>> city.encode('cp437', errors='replace')

b'S?o Paulo'

>>> city.encode('cp437', errors='xmlcharrefreplace')

b'S&#227;o Paulo'

>>>

Coping with UnicodeDecodeError:

t every byte holds a valid ASCII character, and not every byte sequence is valid UTF-8 or UTF-16; therefore, when you assume one of these encodings while converting a binary sequence to text, you will get a UnicodeDecodeError if unexpected bytes are found.

On the other hand, many legacy 8-bit encodings like 'cp1252', 'iso8859\_1', and 'koi8\_r' are able to decode any stream of bytes, including random noise, without generating errors. Therefore, if your program assumes the wrong 8-bit encoding, it will silently decode garbage

Decoding from str to bytes: success and error handling:

Output: X

>>>

>>> octets = b'Montr\xe9al'

>>> octets.decode('cp1252')

'Montréal'

>>> octets.decode('iso8859\_7')

'Montrιal'

>>> octets.decode('koi8\_r')

'MontrИal'

>>> octets.decode('utf\_8')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:

invalid continuation byte

>>> octets.decode('utf\_8', errors='replace')

'Montr�al

>>>

There is a lot happening here. In the first case there is no problem because ‘cp1252’ is a superset of latin1, which is the type of encode that holds that character. The next two cases are intended for different languages, Greek, and Russian so they return garbage characters and the code fails silently, which is the worst case scenario. Finally, we have an error because UTF-8 simply can’t handle those bytes.

SyntaxError When Loading Modules with Unexpected Encoding:

UTF-8 is the default source encoding for Python 3, just as ASCII was the default for Python 2 (starting with 2.5). If you load a .py module containing non-UTF-8 data and no encoding declaration, you get a message like this:

Error raised due to decoding exception:

SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line

1, but no encoding declared; see http://python.org/dev/peps/pep-0263/

for details>>>

Because UTF-8 is widely deployed in GNU/Linux and OSX systems, a likely scenario is opening a .py file created on Windows with cp1252. Note that this error happens even in Python for Windows, because the default encoding for Python 3 is UTF-8 across all platforms.

To fix this problem, add a magic coding comment at the top of the file:

“Hello, World!” in Portuguese:

# coding: cp1252

print(¡’Olá, Mundo!’)

How to Discover the Encoding of a Byte Sequence:

How do you find the encoding of a byte sequence? Short answer: *you can’t*. You must be told.

Some communication protocols and file formats, like HTTP and XML, contain headers that explicitly tell us how the content is encoded. You can be sure that some byte streams are not ASCII because they contain byte values over 127, and the way UTF-8 and UTF-16 are built also limits the possible byte sequences. But even then, you can never be 100% positive that a binary file is ASCII or UTF-8 just because certain bit patterns are not there.

However, considering that human languages also have their rules and restrictions, once you assume that a stream of bytes is human plain text it may be possible to sniff out its encoding using heuristics and statistics. For example, if b'\x00' bytes are common, it is probably a 16- or 32-bit encoding, and not an 8-bit scheme, because null characters in plain text are bugs; when the byte sequence b'\x20\x00' appears often, it is likely to be the space character (U+0020) in a UTF-16LE encoding, rather than the obscure U +2000 EN QUAD character—whatever that is.

BOM: A Useful Gremlin:

In the example from before we saw the decoding of the string ‘El Niño’:

>>> u16 = 'El Niño'.encode('utf\_16')

>>> u16

b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'>>>

The bytes are b'\xff\xfe'. That is a BOM—byte-order mark—denoting the “little-endian” byte ordering of the Intel CPU where the encoding was performed. On a little-endian machine, for each code point the least significant byte comes first: the letter 'E', code point U+0045 (decimal 69), is encoded in byte offsets 2 and 3 as 69 and 0:

>>> list(u16)

[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]

>>>

There is a variant of UTF-16—UTF-16LE—that is explicitly little-endian, and another one explicitly big-endian, UTF-16BE. If you use them, a BOM is not generated:

Output: X

>>>

>>> u16le = 'El Niño'.encode('utf\_16le')

>>> list(u16le)

[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]

>>> u16be = 'El Niño'.encode('utf\_16be')

>>> list(u16be)

[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]

>>>

If present, the BOM is supposed to be filtered by the UTF-16 codec, so that you only get the actual text contents of the file without the leading ZERO WIDTH NO-BREAK SPACE. The standard says that if a file is UTF-16 and has no BOM, it should be assumed to be UTF-16BE (big-endian). However, the Intel x86 architecture is little-endian, so there is plenty of little-endian UTF-16 with no BOM in the wild.

This whole issue of endianness only affects encodings that use words of more than one byte, like UTF-16 and UTF-32. One big advantage of UTF-8 is that it produces the same byte sequence regardless of machine endianness, so no BOM is needed. Nevertheless, some Windows applications (notably Notepad) add the BOM to UTF-8 files anyway— and Excel depends on the BOM to detect a UTF-8 file, otherwise it assumes the content is encoded with a Windows codepage. The character U+FEFF encoded in UTF-8 is the three-byte sequence b'\xef\xbb\xbf'. So if a file starts with those three bytes, it is likely to be a UTF-8 file with a BOM. However, Python does not automatically assume a file is UTF-8 just because it starts with b'\xef\xbb\xbf'.

Handling Text Files

The best practice for handling text is the “Unicode sandwich” This means that bytes should be decoded to str as early as possible on input (e.g., when opening a file for reading). The “meat” of the sandwich is the business logic of your program, where text handling is done exclusively on str objects. You should never be encoding or decoding in the middle of other processing. On output, the str are encoded to bytes as late as possible. Most web frameworks work like that, and we rarely touch bytes when using them. In Django, for example, your views should output Unicode str; Django itself takes care of encoding the response to bytes, using UTF-8 by default.

Unicode sandwich: current best practice for text processing:

Graphical user interface, text, application

Description automatically generated

Python 3 makes it easier to follow the advice of the Unicode sandwich, because the open built-in does the necessary decoding when reading and encoding when writing files in text mode, so all you get from my\_file.read() and pass to my\_file.write(text) are str objects.

Code that has to run on multiple machines or on multiple occasions should never depend on encoding defaults. Always pass an explicit encoding= argument when opening text files, because the default may change from one machine to the next, or from one day to the next.

Encoding bug and how to fix it:

Output: X

>>>

>>> fp = open('cafe.txt', 'w', encoding='utf\_8')

>>> fp

<\_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf\_8'>

>>> fp.write('café')

4

>>> fp.close()

>>> import os

>>> os.stat('cafe.txt').st\_size

5

>>> fp2 = open('cafe.txt')

>>> fp2

<\_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>

>>> fp2.encoding

'cp1252'

>>> fp2.read()

'cafÃ©'

>>> fp3 = open('cafe.txt', encoding='utf\_8')

>>> fp3

<\_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf\_8'>

>>> fp3.read()

'café'

Here we create a file, in which ‘utf-8’ encoding is used explicitly. Saving a text file in memory by assigning a variable to it creates an <\_io.TextIOWrapper> object that holds all that info. When we write the string 'café' on it the output is a number 4, this references the number of bytes written into the file. When we go and check its length with os.stat('cafe.txt').st\_size we get , this is because there are actually bytes on the file. When we open it with the default decoder form windows which is <'cp1252'> we can see that there are actually 5 bytes, but not the right ones because we are reading it with another codec, which is a different one from the one that was used. But if we open the file with the right codec we can see the intended text.

The bytewise :

Output: X

>>> fp4 = open('cafe.txt', 'rb')

>>> fp4

<\_io.BufferedReader name='cafe.txt'>

>>> fp4.read()

b'caf\xc3\xa9'

>>>

Here we can see how looks like when we open a file as bytes. The reason why the three first bytes are read as ‘caf’ is because they fall in the range of ASCII, so they are automatically interpreted as so. While the other two bytes are not in the ASCII range, and what we see is the representation of those bytes in hexadecimal numbers.

Normalizing Unicode for Saner Comparisons NFC and NFD

Normalizing Unicode values means to fix the error from before. There are two types we’ll see first. NFC and NFD. What we are essentially doing here is bringing the accent to its short form in NFC, and to its long form (5 bytes) in NFD:

NFC and NFD Normalization protocols:

Output: X

>>>

>>> from unicodedata import normalize

>>> s1 = 'café' # composed "e" with acute accent

>>> s2 = 'cafe\u0301' # decomposed "e" and acute accent

>>> len(s1), len(s2)

(4, 5)

>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))

(4, 4)

>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))

(5, 5)

>>> normalize('NFC', s1) == normalize('NFC', s2)

True

>>>

Western keyboards usually generate composed characters, so text typed by users will be in NFC by default. However, to be safe, it may be good to sanitize strings with normal ize('NFC', user\_text) before saving.

NFC and NFD Normalization protocols(Single characters to single characters):

Output: X

>>>

>>> from unicodedata import normalize, name

>>> ohm = '\u2126'

>>> name(ohm)

'OHM SIGN'

>>> ohm\_c = normalize('NFC', ohm)

>>> name(ohm\_c)

'GREEK CAPITAL LETTER OMEGA'

>>> ohm == ohm\_c

False

>>> normalize('NFC', ohm) == normalize('NFC', ohm\_c)

True>>>

Normalizing Unicode for Saner Comparisons NFKC and NFKD

In the acronyms for the other two normalization forms—NFKC and NFKD—the letter K stands for “compatibility.” These are stronger forms of normalization, affecting the so-called “compatibility characters.”

Here is how the NFKC works in practice::

Output: X

>>>

>>> from unicodedata import normalize, name

>>> half = '½'

>>> normalize('NFKC', half)

'1⁄2'

>>> four\_squared = '4²'

>>> normalize('NFKC', four\_squared)

'42'

>>> micro = 'µ'

>>> micro\_kc = normalize('NFKC', micro)

>>> micro, micro\_kc

('µ', 'μ')

>>> ord(micro), ord(micro\_kc)

(181, 956)

>>> name(micro), name(micro\_kc)

('MICRO SIGN', 'GREEK SMALL LETTER MU')

>>>

Importance of the K in NFKC

Although '1⁄2' is a reasonable substitute for '½', and the micro sign is really a lowercase Greek mu, converting '4²' to '42' changes the meaning. An application could store '4²' as '42', but the normalize function knows nothing about format‐ ting. Therefore, NFKC or NFKD may lose or distort information, but they can produce convenient intermediate representations for searching and indexing: users may be pleased that a search for '1⁄2 inch' also finds documents containing '½ inch'.

NFKC and NFKD normalization should be applied with care and only in special cases—e.g., search and indexing—and not for permanent storage, because these transformations cause data loss.

Case Folding

Case folding is essentially converting all text to lowercase, with some additional trans‐ formations. It is supported by the str.casefold() method (new in Python 3.3). For any string s containing only latin1 characters, s.casefold() produces the same result as s.lower(), with only two exceptions—the micro sign 'µ' is changed to the Greek lowercase mu (which looks the same in most fonts) and the German Eszett or “sharp s” (ß) becomes “ss”:

Micro sign and Sharp S:

Output: X

>>>

harp s” (ß) becomes “ss”:

>>> micro = 'µ'

>>> name(micro)

'MICRO SIGN'

>>> micro\_cf = micro.casefold()

>>> name(micro\_cf)'GREEK SMALL LETTER MU'

>>> micro, micro\_cf

('µ', 'μ')

>>> eszett = 'ß'

>>> name(eszett)

'LATIN SMALL LETTER SHARP S'

>>> eszett\_cf = eszett.casefold()

>>> eszett, eszett\_cf

('ß', 'ss')>>>

As of Python 3.4, there are 116 code points for which str.casefold() and str.lower() return different results. That’s 0.11% of a total of 110,122 named characters in Unicode 6.3.

Dual-Mode str and bytes APIs

The standard library has functions that accept str or bytes arguments and behave differently depending on the type. Some examples are in the re and os modules.

str Versus bytes in Regular Expressions

:

Output: X

>>>

>>>

:

Output: X

>>>

>>>