What is the difference between a string and a byte string?

Asked 8 years, 11 months ago Active 22 days ago Viewed 121k times

202

I am working with a library which returns a byte string and I need to convert this to a string.

Although I'm not sure what the difference is - if any.

python string character byte

122

edited May 22 '18 at 21:34
Suraj
55 8

asked Jun 3 '11 at 7:06
Sheldon
6,580 16 53 87

7 Answers

Active | Oldest | Votes

251

Assuming Python 3 (in Python 2, this difference is a little less well-defined) - a string is a sequence of characters, ie unicode codepoints; these are an abstract concept, and can't be directly stored on disk. A byte string is a sequence of, unsurprisingly, bytes - things that *can* be stored on disk. The mapping between them is an *encoding* - there are quite a lot of these (and infinitely many are possible) - and you need to know which applies in the particular case in order to do the conversion, since a different encoding may map the same bytes to a different string:

```
>>> b'\xcf\x84o\xcf\x81\xce\xbdo\xcf\x82'.decode('utf-16')
'蘅콯캁濠苏'
>>> b'\xcf\x84o\xcf\x81\xce\xbdo\xcf\x82'.decode('utf-8')
'τορνος'
```

Once you know which one to use, you can use the `.decode()` method of the byte string to get the right character string from it as above. For completeness, the `.encode()` method of a character string goes the opposite way:

```
>>> 'τορνος'.encode('utf-8')
b'\xcf\x84o\xcf\x81\xce\xbdo\xcf\x82'
```

edited May 10 '17 at 9:47

answered Jun 3 '11 at 7:49
lvc
28.6k 5 56 93

7 To clarify for Python 2 users: the `str` type is the same as the `bytes` type; this answer is equivalently comparing the `unicode` type (does not exist in Python 3) to the `str` type. – craymichael Nov 10 '16 at 16:02

To be technically correct, unicode is not the default encoding, rather the `utf-8` encoding is the default character encoding to store unicode strings in memory. – Kshitij Saraogi May 10 '17 at 9:03

2 @KshitijSaraogi that isn't quite true either; that whole sentence was edited in and is a bit unfortunate. The in-memory representation of Python 3 `str` objects is not accessible or relevant from the Python

1 If they can't be directly stored on disk, so how are they stored in memory? – z33k Nov 4 '17 at 14:38

1 @orety they do have to be encoded *somehow* internally for exactly that reason, but this isn't expos3s to you from Python code much like you don't have to care about how floating point numbers are stored. – lvc Nov 5 '17 at 22:43

383

+100

The only thing that a computer can store is bytes.

To store anything in a computer, you must first *encode* it, i.e. convert it to bytes. For example:

- If you want to store music, you must first *encode* it using `MP3`, `WAV`, etc.
- If you want to store a picture, you must first *encode* it using `PNG`, `JPEG`, etc.
- If you want to store text, you must first *encode* it using `ASCII`, `UTF-8`, etc.

`MP3`, `WAV`, `PNG`, `JPEG`, `ASCII` and `UTF-8` are examples of *encodings*. An encoding is a format to represent audio, images, text, etc in bytes.

In Python, a byte string is just that: a sequence of bytes. It isn't human-readable. Under the hood, everything must be converted to a byte string before it can be stored in a computer.

On the other hand, a character string, often just called a "string", is a sequence of characters. It is human-readable. A character string can't be directly stored in a computer, it has to be *encoded* first (converted into a byte string). There are multiple encodings through which a character string can be converted into a byte string, such as `ASCII` and `UTF-8`.

```
'I am a string'.encode('ASCII')
```

The above Python code will encode the string `'I am a string'` using the encoding `ASCII`. The result of the above code will be a byte string. If you print it, Python will represent it as `b'I am a string'`. Remember, however, that byte strings *aren't human-readable*, it's just that Python decodes them from `ASCII` when you print them. In Python, a byte string is represented by a `b`, followed by the byte string's `ASCII` representation.

A byte string can be *decoded* back into a character string, if you know the encoding that was used to encode it.

```
b'I am a string'.decode('ASCII')
```

The above code will return the original string `'I am a string'`.

Encoding and decoding are inverse operations. Everything must be encoded before it can be written to disk, and it must be decoded before it can be read by a human.

edited Jan 7 '17 at 16:48 Mark Amery

answered Jul 9 '15 at 15:46 Zanadix

58 Zenadix deserves some kudos here. After some years functioning in this environment, his is the first explanation that clicked with me. I may tattoo it on my other arm (one arm already has "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) by Joel Spolsky" – neil.millikin Jul 16 '15 at 12:06

4 Absolutely brilliant. Lucid and easy to understand. However, I would like to mention that this line - "If you print it, Python will represent it as b'I am a string'" is true for Python3 as for Python2 bytes and str are the same thing. – SRC Dec 17 '16 at 9:11

5 I am awarding you this bounty for offering a very human-readable explanation to put some clarity in this subject! – fedorqui 'SO stop harming' Jan 8 '17 at 15:08

3 Great answer. The only thing that could perhaps be added is to point out more clearly that historically, programmers and programming languages have tended to explicitly or implicitly *assume that a byte sequence and an ASCII string were the same thing*. Python 3 decided to explicitly break this assumption, correctly IMHO. – nekomatic Jan 17 '17 at 9:39

4 Link to Joel's post mentioned by @neil.millikin above : joelonsoftware.com/2003/10/08/… – Kshitij Saraogi May 7 '17 at 15:54

13

Note: I will elaborate more my answer for Python 3 since the end of life of Python 2 is very close.

In Python 3

`bytes` consists of sequences of 8-bit unsigned values, while `str` consists of sequences of Unicode code points that represent textual characters from human languages.

```
>>> # bytes
>>> b = b'h\x65llo'
>>> type(b)
<class 'bytes'>
>>> list(b)
[104, 101, 108, 108, 111]
>>> print(b)
b'hello'
>>>
>>> # str
>>> s = 'nai\u0308ve'
>>> type(s)
<class 'str'>
>>> list(s)
['n', 'a', 'i', '̈', 'v', 'e']
>>> print(s)
naïve
```

Even though `bytes` and `str` seem to work the same way, their instances are not compatible with each other, i.e, `bytes` and `str` instances can't be used together with operators like `>` and `+`. In addition, keep in mind that comparing `bytes` and `str` instances for equality, i.e. using `==`, will always evaluate to `False` even when they contain exactly the same characters.

```
>>> # concatenation
>>> b'hi' + b'bye' # this is possible
b'hibye'
>>> 'hi' + 'bye' # this is also possible
'hibye'
>>> b'hi' + 'bye' # this will fail
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "bytes") to str
>>>
>>> # comparison
>>> b'red' > b'blue' # this is possible
True
>>> 'red'> 'blue' # this is also possible
True
>>> b'red' > 'blue' # you can't compare bytes with str
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'bytes' and 'str'
>>> 'red' > b'blue' # you can't compare str with bytes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'bytes'
>>> b'blue' == 'red' # equality between str and bytes always evaluates to False
False
>>> b'blue' == 'blue' # equality between str and bytes always evaluates to False
False
```

Another issue when dealing with `bytes` and `str` is present when working with files that are returned using the `open` built-in function. On one hand, if you want ot read or write binary data to/from a file, always open the file using a binary mode like 'rb' or 'wb'. On the other hand, if you want to read or write Unicode data to/from a file, be aware of the default encoding of your computer, so if necessary pass the `encoding` parameter to avoid surprises.

In Python 2

`str` consists of sequences of 8-bit values, while `unicode` consists of sequences of Unicode characters. One thing to keep in mind is that `str` and `unicode` can be used together with operators if `str` only consists of 7-bit ASCI characters.

It might be useful to use helper functions to convert between `str` and `unicode` in Python 2, and between `bytes` and `str` in Python 3.

edited Dec 8 '19 at 17:01

answered Sep 4 '17 at 13:12

lmiguelvargasf
28.2k 23 126 137

4

From What is Unicode:

> Fundamentally, computers just deal with numbers. They store letters and other characters by assigning a number for each one.
>
> .....
>
> Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.

So when a computer represents a string, it finds characters stored in the computer of the string through their unique Unicode number and these figures are stored in memory. But you can't directly write the string to disk or transmit the string on network through their unique Unicode number because these figures are just simple decimal number. You should encode the string to byte string, such as `UTF-8`. `UTF-8` is a character encoding capable of encoding all possible

encoded in `UTF-8` from other systems, your computer will decode it and display characters in it through their unique Unicode number. When a browser receive string data encoded `UTF-8` from network, it will decode the data to string (assume the browser in `UTF-8` encoding) and display the string.

In python3, you can transform string and byte string to each other:

```
>>> print('中文'.encode('utf-8'))
b'\xe4\xb8\xad\xe6\x96\x87'
>>> print(b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8'))
中文
```

In a word, string is for displaying to humans to read on a computer and byte string is for storing to disk and data transmission.

answered Apr 23 '17 at 12:52

Sam Yang
319 3 9

1

Unicode is an agreed-upon format for the binary representation of characters and various kinds of formatting (e.g. lower case/upper case, new line, carriage return), and other "things" (e.g. emojis). A computer is no less capable of storing a unicode representation (a series of bits), whether in memory or in a file, than it is of storing an ascii representation (a different series of bits), or any other representation (series of bits).

For *communication* to take place, the parties to the communication must agree on what representation will be used.

Because unicode seeks to represent *all* the possible characters (and other "things") used in inter-human and inter-computer communication, it requires a greater number of bits for the representation of many characters (or things) than other systems of representation that seek to represent a more limited set of characters/things. To "simplify," and perhaps to accommodate historical usage, unicode representation is almost exclusively converted to some other system of representation (e.g. ascii) for the purpose of storing characters in files.

It is not the case that unicode *cannot* be used for storing characters in files, or transmitting them through *any* communications channel, simply that it *is* not.

The term "string," is not precisely defined. "String," in its common usage, refers to a set of characters/things. In a computer, those characters may be stored in any one of many different bit-by-bit representations. A "byte string" is a set of characters stored using a representation that uses eight bits (eight bits being referred to as a byte). Since, these days, computers use the unicode system (characters represented by a variable number of bytes) to store characters in memory, and byte strings (characters represented by single bytes) to store characters to files, a conversion must be used before characters represented in memory will be moved into storage in files.

answered Oct 3 '19 at 19:09

Gordon Shephard
21 3

0

```
>>> 'š'.encode('utf-8')
b'\xc5\xa1'
```

For the purpose of this example let's display the sequence of bytes in its binary form:

```
>>> bin(int(b'\xc5\xa1'.hex(), 16))
'0b1100010110100001'
```

Now it is **generally not possible** to decode the information back without knowing how it was encoded. Only if you know that the `utf-8` text encoding was used, you can follow the algorithm for decoding utf-8 and acquire the original string:

```
11000101 10100001
   ^^^^^   ^^^^^^
   00101   100001
```

You can display the binary number `101100001` back as a string:

```
>>> chr(int('101100001', 2))
'š'
```

answered Apr 28 at 13:06

Jeyekomon
1,297 1 15 23

0

The Python languages includes `str` and `bytes` as standard "Built-in Types". In other words, they are both classes. I don't think it's worthwhile trying to rationalize why Python has been implemented this way.

Having said that, `str` and `bytes` are very similar to one another. Both share most of the same methods. The following methods are unique to the `str` class:

```
casefold
encode
format
format_map
isdecimal
isidentifier
isnumeric
isprintable
```

The following methods are unique to the `bytes` class:

```
decode
fromhex
hex
```

edited May 6 at 19:39

answered May 6 at 19:34

fiftytwocards
21 1 4