What exactly do “u” and “r” string flags do, and what are raw string literals?

Ask Question

521

179

While asking this question, I realized I didn't know much about raw strings. For somebody claiming to be a Django trainer, this sucks.

I know what an encoding is, and I know what `u''` alone does since I get what is Unicode.

- But what does `r''` do exactly? What kind of string does it result in?
- And above all, what the heck does `ur''` do?
- Finally, is there any reliable way to go back from a Unicode string to a simple raw string?
- Ah, and by the

charset are set to UTF-8, does `u''` actually do anything?

python unicode python-2.x rawstring

edited Oct 10 '18 at 19:54

wim
168k 54 322 455

asked Jan 17 '10 at 16:22

e-satis
364k 98 265 310

6 Answers

580

There's not really any "raw *string*"; there are raw *string literals*, which are exactly the string literals marked by an `'r'` before the opening quote.

A "raw string literal" is a slightly different syntax for a string literal, in which a backslash, `\`, is taken as meaning "just a backslash" (except when it comes right before a quote that would otherwise terminate the literal) -- no "escape sequences" to represent newlines, tabs,

backslash must be doubled up to avoid being taken as the start of an escape sequence.

This syntax variant exists mostly because the syntax of regular expression patterns is heavy with backslashes (but never at the end, so the "except" clause above doesn't matter) and it looks a bit better when you avoid doubling up each of them -- that's all. It also gained some popularity to express native Windows file paths (with backslashes instead of regular slashes like on other platforms), but that's very rarely needed (since normal slashes mostly work fine on Windows too) and imperfect (due to the "except" clause above).

`r'...'` is a byte string (in Python 2.*), `ur'...'` is a Unicode string (again, in Python 2.*), and any of the other three kinds of quoting also produces exactly the same types of strings (so for example `r'...'`

byte strings, and so on).

Not sure what you mean by "going *back*" - there is no intrinsically back and forward directions, because there's no raw string **type**, it's just an alternative syntax to express perfectly normal string objects, byte or unicode as they may be.

And yes, in Python 2.*, `u'...'` **is** of course always distinct from just `'...'` -- the former is a unicode string, the latter is a byte string. What encoding the literal might be expressed in is a completely orthogonal issue.

E.g., consider (Python 2.6):

```
>>> sys.getsizeof
28
>>> sys.getsizeof
34
```

The Unicode object of course takes more memory space (very small difference for a very short string, obviously ;-).

dited Apr 29 '17 at 20:22

Ninjakannon

2,721 4 30 52

637k 129 1046 1286

4 Understanding "r" doesn't implies any type or encoding issues, it's much simpler. – e-satis Jan 17 '10 at 16:42

19 Note that ru"C:\foo\unstable" will fail because \u is a unicode escape sequence in ru mode. r mode does not have \u. – Curtis Yallop Jun 9 '14 at 16:08

21 Note that `u` and `r` are not commutative: `ur'str'` works, `ru'str'` doesnt. (at least in ipython 2.7.2 on win7) – RafiK Jul 10 '14 at 13:21

2 docs.python.org/2/reference/… – k107 Apr 7 '15 at 4:57

4 Just tested `r` strings and noticed that if `\` is the last character it will not be taken as a literal but instead escapes the closing quote, causing `SyntaxError: EOL while scanning string literal`. So `\\` still must be used for the final instance of `\` in any

19 '17 at 14:00

155

There are two types of string in python: the traditional `str` type and the newer `unicode` type. If you type a string literal without the `u` in front you get the old `str` type which stores 8-bit characters, and with the `u` in front you get the newer `unicode` type that can store any Unicode character.

The `r` doesn't change the type at all, it just changes how the string literal is interpreted. Without the `r`, backslashes are treated as escape characters. With the `r`, backslashes are treated as literal. Either way, the type is the same.

`ur` is of course a Unicode string where backslashes are literal backslashes, not part of escape codes.

`str()` function, but if there are any unicode characters that cannot be represented in the old string, you will get an exception. You could replace them with question marks first if you wish, but of course this would cause those characters to be unreadable. It is not recommended to use the `str` type if you want to correctly handle unicode characters.

dited Apr 28 '14 at 18:51

Stefan van den Akker
4,174 6 32 49

nswered Jan 17 '10 at 16:26

Mark Byers
601k 129 1372 1349

Thanks, accepted. As I said, I knaw what unicode is, I didn't know what "r" meant and what would be the combination of "u" and "r". I know better know, cheers. – e-satis Jan 17 '10 at 16:37

6 Backslashes are not treated as literal in raw string literals, which is why `r"\"` is a syntax error. – Roger Pate Jan 17 '10 at 16:38

I'll have to

better. Damn him ! – e-satis Jan 17 '10 at 16:41

simple and precise answer – sandyp Apr 21 '17 at 16:22

2 Only applies to Python 2. – PaulMcG Oct 11 '18 at 15:54

45

'raw string' means it is stored as it appears. For example, `'\'` is just a *backslash* instead of an *escaping*.

dited Feb 26 at 20:30

simhumileco
8,057 3 59 58

nswered Mar 6 '12 at 1:21

xiaolong
1,441 3 18 32

2 short simple and clear :) – Iman Dec 30 '18 at 9:41

31

A "u" prefix denotes the value has type `unicode` rather than `str`.

Raw string literals, with an "r" prefix, escape any escape sequences within them, so `len(r"\n")` is 2. Because they escape escape sequences, you cannot end a string

sequence (e.g. `r"\"`).

"Raw" is not part of the type, it's merely one way to represent the value. For example, `"\\n"` and `r"\n"` are identical values, just like `32` , `0x20` , and `0b100000` are identical.

You can have unicode raw string literals:

```
>>> u = ur"\n"
>>> print type(u)
<type 'unicode'>
```

The source file encoding just determines how to interpret the source file, it doesn't affect expressions or types otherwise. However, it's recommended to avoid code where an encoding other than ASCII would change the meaning:

> Files using ASCII (or UTF-8, for Python 3.0) should not have a coding cookie. Latin-1 (or UTF-8) should only be used when a comment or docstring needs to mention an author name

> escapes is the preferred way to include non-ASCII data in string literals.

dited Jan 17 '10 at 16:55

ıswered Jan 17 '10 at 16:25
Roger Pate

25

Let me explain it simply: In python 2, you can store string in 2 different types.

The first one is **ASCII** which is **str** type in python, it uses 1 byte of memory. (256 characters, will store mostly English alphabets and simple symbols)

The 2nd type is **UNICODE** which is **unicode** type in python, it uses 2 bytes of memory. (65536 characters, so this include all characters of all languages on earth)

By default, python will prefer **str** type but if you want to store string in **unicode** type you can put **u** in front of the text like **u'text'** or you can do this by calling

function to cast **str** to **unicode**. That's it!

Now the **r** part, you put it in front of the text to tell the computer that the text is raw text, backslash should not be an escaping character. **r'\n'** will not create a new line character. It's just plain text containing 2 characters.

If you want to convert **str** to **unicode** and also put raw text in there, use **ur** because **ru** will raise an error.

NOW, the important part:

You cannot store one backslash by using **r**, it's the only exception. So this code will produce error: **r'\'**

To store a backslash (only one) you need to use **'\\'**

If you want to store more than 1 characters you can still use **r** like **r'\\'** will produce 2 backslashes as you expected.

I don't know the reason why **r** doesn't work with one backslash

anyone yet. I hope that it is a bug.

dited Jan 10 '17 at 16:23

nswered Aug 25 '15 at 21:01

off99555
1,030 14 17

9 You will notice not only `r'\'` is illegal, you even can't put a single `'\'` at any string's tail. Just like `r'xxxxxx\'` is a illegal string. – diverger Jun 27 '16 at 6:56

4

Maybe this is obvious, maybe not, but you can make the string **'\'** by calling **x=chr(92)**

```
x=chr(92)
print type(x), le
y='\\'
print type(y), le
x==y   # True
x is y # False
```

nswered May 15 '17 at 7:37

Bomba Ps
59 4

3 `x is y` evaluates to True in python3? – Habeeb Perwad Nov 29 '17 at 3:22

5 @HabeebPerwad, that is because of *string interning*. You

`True` because of interning. Instead use `x == y` (if your not checking if x and y are exactly the same object stored at a single memory position, that is). – Lucubrator Dec 11 '17 at 19:12

protected by codeforester Oct 10 '18 at 19:54

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?