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

Asked 10 years, 10 months ago Active 5 months ago

Viewed 459k times

682

222

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 way, if your system and your text editor 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

246k 73 446 590

asked Jan 17 '10 at 16:22

e-satis
497k 103 281 318

7 Answers

Active | Oldest | Votes

706

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, backspaces, form-feeds, and so on. In normal string literals, each 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'...'`, `r'''...'''`, `r"..."`, `r"""..."""` are all 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('ciao')
28
>>> sys.getsizeof(u'ciao')
34
```

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

edited Apr 29 '17 at 20:22

Ninjakannon
3,178 5 41 61

answered Jan 17 '10 at 16:38

Alex Martelli
732k 150 1135 1328

---

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

---

25 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

---

28 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

---

8 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 strings ending with a backslash. – Enteleform Mar 19 '17 at 14:00

---

1 python 3.x - `sys.getsizeof('cioa') == sys.getsizeof(r'cioa') == sys.getsizeof(u'cioa')` (Ubuntu 16.04 with UTF8 lang). Similarly, `type('cioa') == type(r'cioa') == type(u'cioa')`. BUT, the raw string interpolation makes a difference, so `sys.getsizeof('\ncioa') == sys.getsizeof(u'\ncioa') != sys.getsizeof(r'\ncioa')` – Darren Weber Apr 26 '18 at 15:39

182

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.

You can try to convert a Unicode string to an old string using the `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.

edited Apr 28 '14 at 18:51

Stefan van den Akker

5,395 7 36 55

answered Jan 17 '10 at 16:26

Mark Byers
690k 157 1475 1391

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

---

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

61

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

edited Feb 26 '19 at 20:30

simhumileco
18.7k 10 95 87

answered Mar 6 '12 at 1:21

xiaolong
2,706 3 22 36

---

8 ...unless it's the last character of the string, in which case it does escape the closing quote. – jez Aug 6 '19 at 18:56

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 literal with a single backslash: that's not a valid escape 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), len(u)
<type 'unicode'> 2
```

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 that requires Latin-1; otherwise, using

\x, \u or \U escapes is the preferred way to include non-ASCII data in string literals.

edited Jan 17 '10 at 16:55

answered Jan 17 '10 at 16:25
Roger Pate

31

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. Unicode stores all types of languages.

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 **unicode('text')**

So **u** is just a short way to call a 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 storage but the reason isn't described by anyone yet. I hope that it is a bug.

edited May 29 at 13:33

answered Aug 25 '15 at 21:01

off99555

2,377 23 33

---

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

what about python 3 ? – Krissh Sep 10 '19 at 7:18

2 @Krissh All python 3 strings are Unicode supported. Its type will be `str`. Read more for better understanding here: medium.com/better-programming/… – off99555 Sep 10 '19 at 7:50

5

Unicode string literals

Unicode string literals (string literals prefixed by `u`) are no longer used in Python 3. They are still valid but just for compatibility purposes with Python 2.

Raw string literals

If you want to create a string literal consisting of only easily typable characters like english letters or numbers, you can simply type them: `'hello world'`. But if you want to include also some more exotic characters, you'll have to use some workaround. One of the workarounds are Escape sequences. This way you can for example represent a new line in your string simply by adding two easily typable characters `\n` to your string literal. So when you print the `'hello\nworld'` string, the words will be printed on separate lines. That's very handy!

On the other hand, there are some situations when you want to create a string literal that contains escape sequences but you don't want them to be interpreted by Python. You want them to be **raw**. Look at these examples:

```
'New updates are ready in c:\windows\updates\new'
'In this lesson we will learn what the \n escape se
```

In such situations you can just prefix the string literal with the `r` character like this: `r'hello\nworld'` and no escape sequences will be interpreted by Python. The string will be printed exactly as you created it.

Raw string literals are not completely "raw"?

Many people expect the raw string literals to be raw in a sense that *"anything placed between the quotes is ignored by Python"*. That is not true. Python still recognizes all the escape sequences, it just does not interpret them - it leaves them unchanged instead. It means that **raw string literals still have to be valid string literals**.

From the lexical definition of a string literal:

```
string     ::=  "'" stringitem* "'"
stringitem ::=  stringchar | escapeseq
stringchar ::=  <any source character except "\" or
escapeseq  ::=  "\" <any source character>
```

It is clear that string literals (raw or not) containing a bare quote character: `'hello'world'` or ending with a backslash: `'hello world\'` are not valid.

edited Jul 25 '19 at 12:20

answered Jul 23 '19 at 14:15

Jeyekomon

1,428 1 18 25

4

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

```
x=chr(92)
print type(x), len(x) # <type 'str'> 1
y='\\'
print type(y), len(y) # <type 'str'> 1
x==y   # True
x is y # False
```

answered M

Bom

79

4 `x is y` evaluates to True in pytho

Habeeb Perwad Nov 29 '17 at 3:22

5 @HabeebPerwad, that is because o string interning. You should never rely on the fact that `x is y` happens to evaluate to `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