InfoWorld

by **Serdar Yegulalp**
Senior Writer

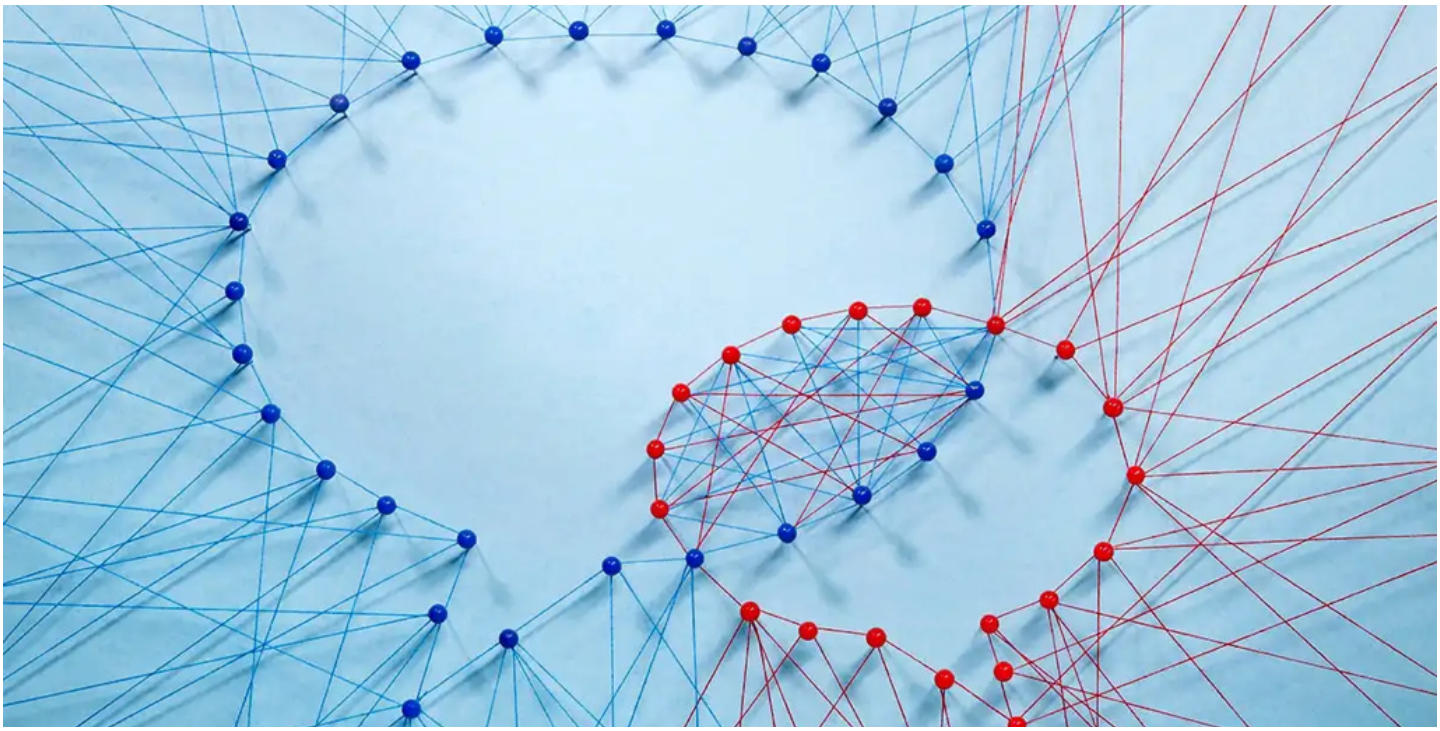
How to use template strings in Python 3.14

How-To

May 14, 2025 • 6 mins

Programming Languages Python Software Development

Python's new template strings, or t-strings, give you a much more powerful way to format data than the old-fashioned f-strings.



Credit: Thinkstock

The familiar formatted string, or f-string, feature in **Python [https://www.infoworld.com/article/2253770/what-is-python-powerful-intuitive-programming.html]** provides a convenient way to print variables as part of a string. But it doesn't let you interact with the way strings are created from those variables. It's hard, if not impossible, to do things like inspect each variable as it's inserted into the string and take some action based on what it is.

Python 3.14 has a new feature called **the template string, or t-string, type [https://peps.python.org/pep-0750/]**. A t-string superficially resembles an f-string, but it's designed to do something very different. Instead of just printing the string with the variables interpolated into it, a t-string lets you process both the string elements and the interpolated variables as separate components, and construct whatever kind of string you like from that.

A basic template string

Template strings look a lot like f-strings. The only difference is the use of `t` instead of `f` to define the string:

```
ame = "Davis"
template = t"Hello, {name}"
```

n

If this were a conventional f-string, we could `print(template)` and get `Hello, Davis` as the output. But if we try printing a t-string, we don't get a string. Instead, we get a Python object representation:

```
Template(strings=('Hello ', ''), interpolations=(Interpolation('Davis',
```

```
    'name', None, ''),))
```

This is because t-strings aren't meant to be printed directly. Instead, we do something with their contents by applying a function or class method.

A t-string object contains different sub-objects we can iterate through, which represent the contents of the t-string:

- The `strings` object contains all the static strings in the original template—in this case, the `"Hello, "` before the variable and the empty string `""` after the variable.
- The `interpolations` object contains all the interpolations of the different variables in the t-string. Each is a separate `Interpolation` object with data about its value, the expression used to create it, and other useful details.

If you iterate through the template directly (`for item in template:`) you would get each element in the template in turn: the string `"Hello, "` , the `Interpolation` object shown above, and the string `""` . This makes it easy to assemble a new string from the elements of a t-string.

Ask a question

ASK

Using a template string

Again, the point of a template string isn't to print it as is, but to pass it to a function that will handle formatting duties. For instance, a simple template string handler to render each variable's string representation in upper case might look like this:

```
from string.templatelib import Template, Interpolation
def t_upper(template: Template):
    output = []
    for item in template:
        if isinstance(item, Interpolation):
            output.append(str(item).upper())
        else:
            output.append(item)
    return "".join(output)
```

If we ran `t_upper()` on the above template, we would get `Hello, DAVIS` as the output.

Note that we have an import line in this code:

```
from string.templatelib import Template, Interpolation
```

`string.templatelib` is a new module in the standard library for Python 3.14 that holds the types we need: `Template` for the type hint to the function, and `Interpolation` to check the elements we iterate through.

However, we don't need `string.templatelib` to make template strings. Those we can construct just by using the t-string syntax anywhere in our code.

A useful template string example

For a better example of how useful and powerful template strings can be, let's look at a task that would be a good deal more difficult without them.
You're probably familiar with HTML sanitization, which removes potentially harmful elements from HTML content so that only safe HTML gets displayed on the web page. Many template engines exist to do this. We can accomplish the same with template strings and more besides:

```
from string.templatelib import Template, Interpolation
import urllib.parse
import html

def clean(input:Template):
    output = []
    inside_tag = False
    for item in input:
        if isinstance(item, Interpolation):
            escape = urllib.parse.quote if inside_tag else html.escape
            out_item = escape(str(item.value))
        else:
            for l in item:
                if l in (""):
                    inside_tag = not inside_tag
            out_item = item
        output.append(out_item)
    return "".join(output)

name=""
print(clean(t'Hello, {name}'))
```

The function `clean()` takes in a template string and cleans it in such a way that any HTML brackets (`<` and `>`) will be changed to harmless literals (`<` and `>`). The surrounding strings in the template string aren't modified.
To do this, the `clean()` function performs three different kinds of processing:

1. If the function encounters a conventional string, which includes HTML we *do* want to render, that string is added to the output as is. It also keeps track of whether or not we are, at that point, inside an HTML tag. (The algorithm used for this isn't very

sophisticated; this is just an example.)

2. If the function encounters an interpolation, and we're inside an HTML tag (for instance, in the `<a href="/user/{name}">` portion of the template), it uses `urllib.parse.quote` to escape the string. This ensures that the string can be included between quotes inside a tag without mangling anything, *and* that the string is URL-safe.

3. If the function encounters an interpolation outside an HTML tag, it uses a different function to escape it (`html.escape`). This also escapes HTML entities but handles quotes differently, and doesn't bother ensuring the output is URL-safe (since we don't need URL safety outside of tags).

Run this code and you'll get the following output:

```
Hello, <David's Friend>
```

Note how spaces in the URL version of the string are escaped, but spaces in the text itself are not. This kind of granular processing of each variable cannot be done easily in an f-string, if at all.

The advantage of t-strings

One big advantage to using t-strings: You can pass the t-string to *different processing functions*. The same t-string can be used to generate different kinds of output simply by changing the function applied. This way, creating the t-string and rendering it to text can be completely separate operations. At the same time, the rendering can be done with full access to the original input, including details about each interpolated variable.

Secure AI by Design: Unleash the power of AI and keep applications, usage and data secure.

Empower your cybersecurity team with expert insights from Palo Alto Networks.
