Positional formatting

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data scientist

datacamp

What is string formatting?

- *String interpolation*
- Insert a custom string or variable in predefined text:

```
custom_string = "String formatting"
print(f"{custom_string} is a powerful technique")
```

```
String formatting is a powerful technique
```

- Usage:
  - Title in a graph
  - Show message or error
  - Pass statement a to function

Methods for formatting

- Positional formatting
- Formatted string literals
- Template method

Positional formatting

- Placeholder replace by value

'text {}'.format(value)

- `str.format()`

```
print("Machine learning provides {} the ability to learn {}".format("systems", "automatically"))
```

```
Machine learning provides systems the ability to learn automatically
```

Positional formatting

- Use variables for both the initial string and the values passed into the method

```
my_string = "{} rely on {} datasets"
method = "Supervised algorithms"
condition = "labeled"
```

```
print(my_string.format(method, condition))
```

```
Supervised algorithms rely on labeled datasets
```

Reordering values

- Include an index number into the placeholders to reorder values

```
print("{} has a friend called {} and a sister called {}".format("Betty", "Linda", "Daisy"))
```

```
Betty has a friend called Linda and a sister called Daisy
```

```
print("{2} has a friend called {0} and a sister called {1}".format("Betty", "Linda", "Daisy"))
```

```
Daisy has a friend called Betty and a sister called Linda
```

Named placeholders

- Specify a name for the placeholders

```
tool="Unsupervised algorithms"
goal="patterns"
print("{title} try to find {aim} in the dataset".format(title=tool, aim=goal))
```

```
Unsupervised algorithms try to find patterns in the dataset
```

Named placeholders

```
my_methods = {"tool": "Unsupervised algorithms", "goal": "patterns"}
```

```
print('{data[tool]} try to find {data[goal]} in the dataset'.format(data=my_methods))
```

```
Unsupervised algorithms try to find patterns in the dataset
```

7. Named placeholders
We can also introduce keyword arguments that are called by their keyword name. In the example code, we inserted keywords in the placeholders. Then, we call these keywords in the format method. We then assign which variable will be passed for each of them resulting in the following output.

8. Named placeholders
Let's examine this code. We have defined a dictionary with keys: tool and goal. We want to insert their values in a string. Inside the placeholders, we can specify the value associated with the key tool of the variable data using bracket notation. Pay attention to the code. Data is the dictionary specified in the method and tool is the key present in that dictionary. So, we get the desired output shown in the slide. Be careful! You need to specify the index without using quotes.

Format specifier

- Specify data type to be used: `{index:specifier}`

```
print("Only {0:f}% of the {1} produced worldwide is {2}!".format(0.5155675, "data", "analyzed"))
```

```
Only 0.515568% of the data produced worldwide is analyzed!
```

```
print("Only {0:.2f}% of the {1} produced worldwide is {2}!".format(0.5155675, "data", "analyzed"))
```

```
Only 0.52% of the data produced worldwide is analyzed!
```

Formatting datetime

```
from datetime import datetime
print(datetime.now())
```

```
datetime.datetime(2019, 4, 11, 20, 19, 22, 58582)
```

```
print("Today's date is {:%Y-%m-%d %H:%M}".format(datetime.now()))
```

```
Today's date is 2019-04-11 20:20
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Formatted string literal

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

f-strings

- Minimal syntax
- Add prefix `f` to string

f"literal string {expression}"

```
way = "code"
method = "learning Python faster"
print(f"Practicing how to {way} is the best method for {method}")
```

```
Practicing how to code is the best method for learning Python faster
```

2. f-strings
The strings defined by this method are called f-strings. They have a minimal syntax as you can see in the slide. To defined them, you need to add the prefix f before the string. Inside quotes, you put your text along with curly braces, which identify placeholders where you insert the expressions.

Type conversion

- Allowed conversions:
  - `!s` (string version)
  - `!r` (string containing a printable representation, i.e. with quotes)
  - `!a` (some as !r but escape the non-ASCII characters)

```
name = "Python"
print(f"Python is called {name!r} due to a comedy series")
```

```
Python is called 'Python' due to a comedy series
```

3. Type conversion
f-strings allow us to convert expressions into different types. We can use exclamation mark s for strings, r for printable representation of strings, or a to escape non-ascii characters. Let's imagine we define the variable name as you can see in the slide. The variable should be surrounded by quotes in the resulting string. We can add after the variable the exclamation r conversion. This will return a printable representation of the string. In the output, we can see that quotes are surrounding the variable.

Format specifiers

- Standard format specifier:
  - `e` (scientific notation, e.g. 5 10^3)
  - `d` (digit, e.g. 4)
  - `f` (float, e.g. 4.5353)

4. Format specifiers
We can also use format specifiers such as e for scientific notation, d for digit and f for float. In the example code, we define a variable containing a number. Then, we insert it in the f-string. We specify that we want it to have only two decimals. And we get the following string.

```
number = 90.41890417471841
print(f"In the last 2 years, {number:.2f}% of the data was produced worldwide!")
```

```
In the last 2 years, 90.42% of the data was produced worldwide!
```

Format specifiers

- `datetime`

5. Format specifiers
We can also format datetime. We only need to insert the variable containing the datetime object. After that, we placed a colon and the specifiers month name, day and year. And we get the string containing the date as we see on the code.

```
from datetime import datetime
my_today = datetime.now()
```

```
print(f"Today's date is {my_today:%B %d, %Y}")
```

```
Today's date is April 14, 2019
```

Index lookups

```
family = {"dad": "John", "siblings": "Peter"}
```

```
print("Is your dad called {family[dad]}?".format(family=family))
```

```
Is your dad called John?
```

- Use quotes for index lookups: `family["dad"]`

```
print(f"Is your dad called {family[dad]}?")
```

```
NameError: name 'dad' is not defined
```

6. Index lookups
Do you remember when we accessed dictionaries from the string format method? To insert the value associated with a specific key, we specify the index without quotes. Let's try the same to access dictionaries in f-strings. As we see in the code, Python raises an error telling us that it cannot find the variable. This is due to the fact we need to surround the index with quotes.

Escape sequences

- Escape sequences: backslashes `\`

```
print("My dad is called "John"")
```

```
SyntaxError: invalid syntax
```

```
my_string = "My dad is called \"John\""
```

```
My dad is called "John"
```

7. Escape sequences
In the first video, we said that a string is anything appearing inside quotes. If the string contains quotes as well, one of the ways to escape the error raised is to add a backslash before the quotes.

Escape sequences

```
family = {"dad": "John", "siblings": "Peter"}
```

- Backslashes are not allowed in f-strings

```
print(f"Is your dad called {family[\"dad\"]}?")
```

```
SyntaxError: f-string expression part cannot include a backslash
```

8. Escape sequences
What does this have to do with f-strings? We said that indexes required quotes. f-strings follow the same rule as strings. If Python finds a second quote, it understands that this is a closing mark. So we could escape the quotes by adding backslash. But, we can observe in the code that backslashes are not allowed in the f-string expression. So our only solution is to use single quotes to get the desire output as we can observe in the slide.

```
print(f"Is your dad called {family['dad']}?")
```

```
Is your dad called John?
```

Inline operations

- Advantage: evaluate expressions and call functions inline

```
my_number = 4
my_multiplier = 7
```

```
print(f'{my_number} multiplied by {my_multiplier} is {my_number * my_multiplier}')
```

```
4 multiplied by 7 is 28
```

9. Inline operations
One of the biggest advantages of f-strings is that they allow us to perform inline operations. In the example, we define two numeric variables. We then insert them into the f-string. We can also multiply them inside the expression. And we get the result of that operation in the output.

Calling functions

```
def my_function(a, b):
  return a + b
```

```
print(f"If you sum up 10 and 20 the result is {my_function(10, 20)}")
```

```
If you sum up 10 and 20 the result is 30
```

10. Calling functions
We can also call functions inside the expression of f-strings. In the example, we define a function. Then, we call this function and pass two numbers inside the expression in the f-string. This will return the value in the final string as we can observe in the slide.

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Template method

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Template strings

- Simpler syntax
- Slower than f-strings
- Limited: don't allow format specifiers
- Good when working with externally formatted strings

Basic syntax

```
from string import Template
my_string = Template('Data science has been called $identifier')
my_string.substitute(identifier="sexiest job of the 21st century")
```

```
'Data science has been called sexiest job of the 21st century'
```

3. Basic syntax
First, you need to create the template string. For that, you use the Template constructor that takes only the string, as you can observe in the slide. Template strings use dollar signs to identify placeholders or identifiers. Then, you need to call the method that substitutes the identifier by the string values. For that, you use the identifier name equal the replacement string.

Substitution

- Use many `$identifier`
- Use variables

```
from string import Template
job = "Data science"
name = "sexiest job of the 21st century"
my_string = Template('$title has been called $description')
my_string.substitute(title=job, description=name)
```

```
'Data science has been called sexiest job of the 21st century'
```

4. Substitution
We can place many identifiers as well as variables when using Template strings. In the example code, we define two variables containing strings. We can create a template having two identifiers with a designated name. Afterward, we call the method substitute to assign the identifiers to the different variables. And we get the following output.

Substitution

- Use `${identifier}` when valid characters follow identifier

```
my_string = Template('I find Python very ${noun}ing but my sister has lost $noun')
my_string.substitute(noun="interest")
```

```
'I find Python very interesting but my sister has lost interest'
```

5. Substitution
Sometimes we need to add extra curly braces after the dollar sign to enclose the identifier. This is required when valid characters follow the identifier but are not part of it. In the example, we need to add the ending -ing immediately after the first identifier. We need to include curly braces. If we don't do it, Python believes that -ing belong to the identifier name. We replace it by the variable noun obtaining the shown output.

Substitution

- Use `$$` to escape the dollar sign

```
my_string = Template('I paid for the Python course only $$ $price, amazing!')
my_string.substitute(price="12.50")
```

```
'I paid for the Python course only $ 12.50, amazing!'
```

6. Substitution
Let's imagine now that you are working with numbers and you want to include the dollar sign as part of a string. Because they are use for identifiers, you will need to escape this character by adding an extra dollar sign. And get the correct output as seen in the code.

Substitution

- Raise error when placeholder is missing

```
favorite = dict(flavor="chocolate")
my_string = Template('I love $flavor $cake very much')
my_string.substitute(favorite)
```

```
Traceback (most recent call last):
KeyError: 'cake'
```

7. Substitution
In the example code, we have defined a dictionary with only one key.
However, when we define our template string, we include two identifiers.
What would happen if we pass this dictionary to the method substitute?
Python will raise an error. It tries to replace every placeholder and some of them are missing.

Substitution

```
favorite = dict(flavor="chocolate")
my_string = Template('I love $flavor $cake very much')
```

```
try:
    my_string.substitute(favorite)
except KeyError:
      print("missing information")
```

```
missing information
```

8. Substitution
We could try using the try except block again. In the slide, you can observe again the syntax. The try part will test the given code. If any error appears the except part will be executed obtaining the following output as a result.

Safe substitution

- Always tries to return a usable string
- Missing placeholders will appear in resulting string

```
favorite = dict(flavor="chocolate")
my_string = Template('I love $flavor $cake very much')
my_string.safe_substitute(favorite)
```

```
'I love chocolate $cake very much'
```

9. Safe substitution
A better way to handle this situation is using the safe substitute method. This method will always try to return a usable string. How? It will place missing placeholders in the resulting string. Let's say we have the same situation as before. Now, if we pass the dictionary to the safe substitute, we will not get an error. Instead, we'll get the identifier dollar sign cake in our resulting string, as you can observe in the output.

Which should I use?

- `str.format()` :
  - Good to start with. Concepts apply to f-strings.
  - Compatible with all versions of Python.
- **f-strings:**
  - Always advisable above all methods.
  - Not suitable if not working with modern versions of Python (3.6+).
- **Template strings:**
  - When working with external or user-provided strings

10. Which should I use?
In summary, how do you decide which string formatting you should use? String format is easy to use. You could start mastering this method and then apply the concepts to f-strings. Moreover, the f-strings are always advisable above all methods. But only if you are working with modern versions of Python. Use template strings only when working with user-provided strings.

Let's practice!

REGULAR EXPRESSIONS IN PYTHON