Google

Reading: String formatting and regular expressions

As you've learned, strings are a crucial class of data because they represent textual information. Data professionals encounter strings all the time, so it's important to become familiar with different ways of manipulating and working with them. This reading will review the string formatting techniques you've learned, and also introduce you to regular expressions.

String formatting

String formatting uses the **format()** method, which belongs to the string class. This method formats and inserts specific substrings into designated places within a larger string. It's useful when you have reusable template text into which you want to insert specific changeable values, for example. The **format()** method is also useful when assigning the strings used to label charts and graphs you make.

Here's an example:

```
x = 'values'
y = 100

print('''String formatting lets you insert {} into strings.
They can even be numbers, like {}.'''.format(x, y))
```

Output:

```
String formatting lets you insert values into strings.
They can even be numbers, like 100.
```

Notice the syntax. The **format()** function inserts its arguments into the braces within the string that it's attached to. The order of insertion follows the order of the arguments. Also, this example includes a helpful trick. Sometimes you'll encounter a very long string. Many editors will allow the string to keep extending to the right on a single line. This is impractical unless you have a very wide monitor, but 79 characters is a conventional maximum length for a single line of Python code. Enclosing your string in triple quotes lets you break the string over multiple lines.

The **format()** function can also insert values into braces using explicitly assigned keyword names, which allow you to mix up the order of the function's arguments without changing the order of their insertion into the final string.

For example:

```
var_a = 'A'
var_b = 'B'
print('{a}, {b}'.format(b=var_b, a=var_a))
```

Output:

```
A, B
```

Because the arguments were named, it didn't matter that they were entered with **var_b** first and **var_a** last; they still were inserted into the string in the order specified.

You can also include the arguments' index numbers within the braces to indicate which arguments get inserted in specific spots:

```
var_a = 'A'
var_b = 'B'
print('{1}, {0}'.format(var_a, var_b))
print('{0}, {1}'.format(var_a, var_b))
```

Output:

```
B, A
A, B
```

You can have as many arguments as you want:

```
print('{}, {}, {}, {}, {}, {} ...'.format(1, 2, 3, 4, 5, 6))
```

Output:

```
1, 2, 3, 4, 5, 6 ...
```

And you can repeat arguments' indices:

```
print('{0}{1}{0}'.format('abra', 'cad'))
```

Output:

```
abracadabra
```

The string **format()** method is a versatile and convenient way to take values that are stored in different variables and insert them into a string.

Literal string interpolation (f-strings)

Another string formatting technique that you'll often encounter when using Python version 3.6+ is literal string interpolation, also known as f-strings. F-strings further minimize the syntax required to embed expressions into strings. They're called f-strings because the expressions always begin with f (or F—they're the same).

For example:

```
var_a = 1
var_b = 2
print(f'{var_a} + {var_b}')
print(f'{var_a + var_b}')
print(f'var_a = {var_a} \nvar_b = {var_b}')
```

Output:

```
1 + 2
3
var_a = 1
var_b = 2
```

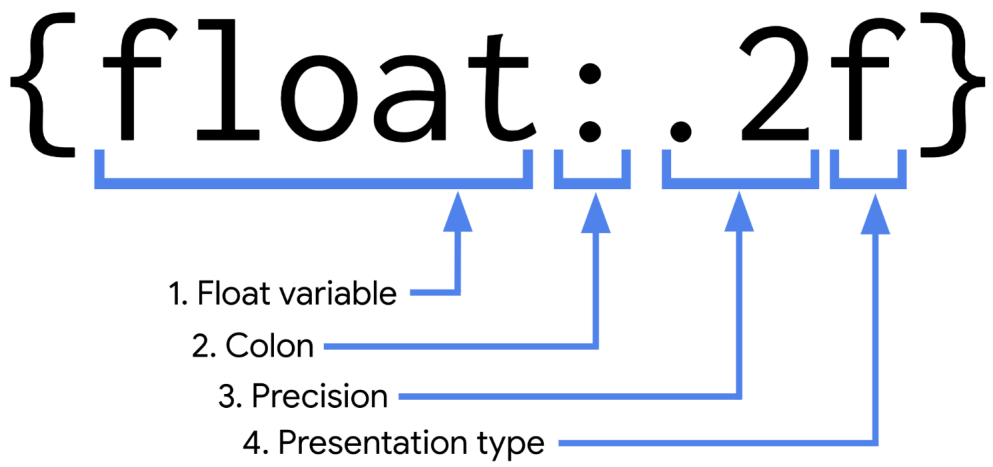
In these examples, the braces still function as the way to indicate where values should be inserted into the string, but they allow you to make the insertion directly, without having to call the **format()** method.

Float formatting options

In addition to inserting expressions into strings, string formatting can format their appearance. There are too many options to list here, but the Python string documentation is a good place to review these techniques. Here are some of the most useful.

To use these options, build your expression within braces as follows.

1. The float variable is what's being formatted
2. A colon (**:**) separates what's being formatted from the syntax used to format it
3. **. number** indicates the desired precision
4. A letter indicates the presentation type



Example:

```
num = 1000.987123
f'{num:.2f}'
```

Output:

```
1000.99
```

This example uses the **f** presentation type to specify that the number contained in the **num** variable should be rounded to two places beyond the decimal.

Type	Meaning
'e'	Scientific notation. For a given precision **p**, formats the number in scientific notation with the letter **'e'** separating the coefficient from the exponent. The coefficient has one digit before and p digits after the decimal point, for a total of **p + 1** significant digits. With no precision given, **e** uses a precision of **6** digits after the decimal point for float, and shows all coefficient digits for decimal.
'f'	Fixed-point notation. For a given precision **p**, formats the number as a decimal number with exactly **p** digits following the decimal point.
'%'	Percentage. Multiplies the number by 100 and displays in fixed (**'f'**) format, followed by a percent sign.

Here are some examples:

```
num = 1000.987123
print(f'{num:.3e}')

decimal = 0.2497856
print(f'{decimal:.4%}')
```

Output:

```
1.001e+03
24.9786%
```

String methods

As one of the primary object classes in Python, strings have many built-in methods designed to facilitate working with them. There are too many of these methods to cover all of them here in depth, but some of the most useful include:

str.count(sub[, start[, end]])

Return the number of non-overlapping occurrences of substring **sub** in the range **[start , end]**.

```
my_string = 'Happy birthday'

print(my_string.count('y'))
print(my_string.count('y', 2, 7))
```

Output:

```
2
1
```

str.find(sub)

Return the lowest index in the string where substring **sub** is found. Return -1 if **sub** is not found.

```
my_string = 'Happy birthday'

my_string.find('birth')
```

Output:

```
6
```

str.join()

Return a string which is the concatenation of the strings in iterable. The separator between elements is the string providing this method.

```
separator_string = ' '
iterable_of_strings = ['Happy', 'birthday', 'to', 'you']

separator_string.join(iterable_of_strings)
```

Output:

```
Happy birthday to you
```

str.partition(sep)

Split the string at the first occurrence of **sep** , and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

```
my_string = 'https://www.google.com/'

my_string.partition('.')
```

Output:

```
('https://www', '.', 'google.com/')
```

str.replace(old, new[, count])

Return a copy of the string with all occurrences of substring **old** replaced by **new**. If the optional argument **count** is given, only the first **count** occurrences are replaced.

```
my_string = 'https://www.google.com/'

my_string.replace('google', 'youtube')
```

Output:

```
https://www.youtube.com/
```

str.split([sep])

Return a list of the words in the string, using **sep** (optional) as the delimiter string. If no **sep** is given, whitespace characters are used as the delimiter. Any number of consecutive whitespaces would indicate a split point, so **' '** (a single whitespace) would split the same way as **' '** (two or more whitespaces).

```
my_string = 'Do you know the muffin man?'

my_string.split()
```

Output:

```
['Do', 'you', 'know', 'the', 'muffin', 'man?']
```

Note that some of these methods have additional optional parameters. This reading covers only the most rudimentary ones. Reference the full string methods documentation for more information on these functions and other methods not included here.

Regular expressions

Regular expressions, also known as regex, refer to techniques that advanced data professionals use to modify and process string data. This program will not require you to use regular expressions in your work, but it's important for you to be aware of the concept. As always, you're encouraged to explore regular expressions on your own.

Regex works by matching patterns in Python. It allows you to search for specific patterns of text within a string of text. Regex is used extensively in web scraping, text processing and cleaning, and data analysis.

The first step in working with regular expressions is to import the **re** module. This module provides the tools necessary for working with regular expressions. Once you have imported the module, you can start working with regular expressions.

Note: The following code block is not interactive.

The basic syntax for a regular expression is:

```
import re
```

```
pattern = 'regex_pattern'

match = re.search(pattern, string)
```

Here is a basic example:

```
import re

my_string = 'Three sad tigers swallowed wheat in a wheat field'

re.search('wall', my_string)
```

Output:

```
<_sre.SRE_Match object; span=(18, 22), match='wall'>
```

This example returns a match object that contains information about the search. In this case, it tells you that the substring **'wall'** does occur in the string from indices 18–22.

Regex is especially useful because it allows you a very high degree of customization when performing your searches.

Here's another example:

```
import re

my_string = 'Three sad tigers swallowed wheat in a wheat field'

re.search('[bms]ad', my_string)
```

Output:

```
<_sre.SRE_Match object; span=(6, 9), match='sad'>
```

This example will search for "bad," "mad," and "sad." Again, these are very basic examples.

Regex has a large catalogue of special expressions that let you search for substrings that will only match if, for example, they are followed by certain characters, or if they don't contain a certain set of characters. It can get very complex. Depending on the work you do as a data professional, you may find yourself exploring regular expressions to analyze and process your data.

Key takeaways

String formatting is the process of inserting specific substrings into designated places within a larger string. Often, the inserted substrings get processed and formatted a certain way. There are multiple ways of using string formatting to help you process strings. These include the **format()** method, literal string interpolations—or f-strings—and regular expressions, also known as regex. The methods you use will depend on what your data demands and your own personal preferences, but it's important to be familiar with the most common techniques used by data professionals.
