## Further on Strings

## String Format Operation

In grade school quizzes a common convention is to use fill-in-the blanks. For instance,

Hello \_\_\_\_\_!

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. We use this as an analogy: Python has a similar construction, better called fill-in-the-braces. There is a particular operation on strings called format, that makes substitutions into places enclosed in braces. For instance the example file, hello\_you3.py, creates and prints the same string as in hello\_you2.py from the previous section:

*'''Hello to you! Illustrates format with {} in print.*

*'''*

person = input('Enter your name: ')

greeting = 'Hello, {}!'.format(person)

print(greeting)

There are several new ideas here!

First method calling syntax for objects is used. You will see this very important modern syntax in more detail at the beginning of the next chapter in[Object Orientation](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/strings3.html#object-orientation). All data in Python are objects, including strings. Objects have a special syntax for functions, called methods, associated with theparticular type of object. In particular str objects have a method called format. The syntax for methods has the object followed by a period followed by the method name, and further parameters in parentheses.

object.methodname(parameters)

In the example above, the object is the string 'Hello {}!'. The method is named format. There is one further parameter, person.

The string for the format method has a special form, with braces embedded. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the format method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier greeting, and then the string is printed.

The identifier greeting was introduced to break the operations into a clearer sequence of steps. However, since the value of greeting is only referenced once, it can be eliminated with the more concise version:

person = input('Enter your name: ')

print('Hello {}!'.format(person))

Consider the interview program. Suppose we want to add a period at the end of the sentence (with no space before it). One approach would be to combine everything with plus signs. Another way is printing with keyword sep=''. Another approach is with string formatting. Using our grade school analogy, the idea is to fill in the blanks in

\_\_\_\_\_ will interview \_\_\_\_\_ at \_\_\_\_\_.

There are multiple places to substitute, and the format approach can be extended to multiple substitutions: Each place in the format string where there is'{}', the format operation will substitute the value of the next parameter in the format parameter list.

Run the example file interview2.py, and check that the results from all three methods match.

*'''Compare print with concatenation and with format string.'''*

applicant = input("Enter the applicant's name: ")

interviewer = input("Enter the interviewer's name: ")

time = input("Enter the appointment time: ")

print(interviewer + ' will interview ' + applicant + ' at ' + time +'.')

print(interviewer, ' will interview ', applicant, ' at ', time, '.', sep='')

print('{} will interview {} at {}.'.format(interviewer, applicant, time))

Sometimes you want a single string, but not just for printing. You can combine pieces with the + operator, but then all pieces must be strings or explicitly converted to strings. An advantage of the format method is that it will convert types to string automatically, like the print function. Here is another variant of our addition sentence example, addition4a.py, using the format method.

*'''Two numeric inputs, explicit sum'''*

x = int(input("Enter an integer: "))

y = int(input("Enter another integer: "))

sum = x+y

sentence = 'The sum of {} and {} is {}.'.format(x, y, sum)

print(sentence)

Conversion to strings was not needed in interview2.py. (Everything started out as a string.) In addition4a.py, however, the automatic conversion of the integers to strings is useful.

So far there is no situation that requires a format string instead of using other approaches. Sometimes a format string provides a shorter and simpler expression. Except where specifically instructed in an exercise for practice, use whatever approach to combining strings and data that you like. There are many elaborations to the fields in braces to control formatting. We will look at one later, [String Formats for Float Precision](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/float.html#precision-formats), where format strings are particularly useful.

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final formatted string. The rule is to double the braces: '{{' and '}}'. The example code formatBraces.py, shown below, makes setStr refer to the string'The set is {5,9}.'. The initial and final doubled braces in the format string generate literal braces in the formatted string:

*'''Illustrate braces in a formatted string.'''*

a = 5

b = 9

setStr = 'The set is {{{}, {}}}.'.format(a, b)

print(setStr)

This kind of format string depends directly on the order of the parameters to the format method. There is another approach with a dictionary, that was used in the first sample program, madlib.py, and will be discussed more in [Dictionaries and String Formatting](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/dictionaries.html#dictionaries-and-string). The dictionary approach is probably the best in many cases, but the count-based approach is an easier start, particularly if the parameters are just used once, in order.

Optional elaboration with explicitly numbered entries

Imagine the format parameters numbered in order, starting from 0. In this case 0, 1, and 2. The number of the parameter position may be included inside the braces, so an alternative to the last line of interview2.py is (added in example file interview3.py):

print('{0} will interview {1} at {2}.'.format(interviewer, applicant, time))

This is more verbose than the previous version, with no obvious advantage. However, if you desire to use some of the parameters more than once, then the approach with the numerical identification with the parameters is useful. Every place the string includes '{0}', the format operation will substitute the value of the initial parameter in the list. Wherever '{1}' appears, the next format parameter will be substituted....

Predict the results of the example file arith.py shown below, if you enter 5 and 6. Then check yourself by running it. In this case the numbers referring to the parameter positions are necessary. They are both repeated and used out of order:

*'''Fancier format string example with*

*parameter identification numbers*

*-- useful when some parameters are used several times.'''*

x = int(input('Enter an integer: '))

y = int(input('Enter another integer: '))

formatStr = '{0} + {1} = {2}; {0} \* {1} = {3}.'

equations = formatStr.format(x, y, x+y, x\*y)

print(equations)

### Short String Exercise

Write a program short.py with a function printShort with heading:

**def** printShort(strings):

*'''Given a list of strings,*

*print the ones with at most three characters.*

*>>> printShort(['a', 'long', one'])*

*a*

*one*

*'''*

In your main program, test the function, calling it several times with different lists of strings. Hint: Find the length of each string with the len function.

The function documentation here models a common approach: illustrating the behavior of the function with a Python Shell interaction. This begins with a line starting with >>>. Other exercises and examples will also document behavior in the Shell.