1.9. Input and Output

We often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python does have a way to create dialog boxes, there is a much simpler function that we can use. Python provides us with a function that allows us to ask a user to enter some data and returns a reference to the data in the form of a string. The function is called `input`.

Python's input function takes a single parameter that is a string. This string is often called the **prompt** because it contains some helpful text prompting the user to enter something. For example, you might call input as follows:

```
aName = input('Please enter your name: ')
```

Now whatever the user types after the prompt will be stored in the `aName` variable. Using the input function, we can easily write instructions that will prompt the user to enter data and then incorporate that data into further processing. For example, in the following two statements, the first asks the user for their name and the second prints the result of some simple processing based on the string that is provided.

Run Load History Show CodeLens

```
1 aName = input("Please enter your name ")
2 print("Your name in all capitals is",aName.upper(),
3       "and has length", len(aName))
4
```

ActiveCode: 1 The input Function Returns a String (strstuff)

It is important to note that the value returned from the `input` function will be a string representing the exact characters that were entered after the prompt. If you want this string interpreted as another type, you must provide the type conversion explicitly. In the statements below, the string that is entered by the user is converted to a float so that it can be used in further arithmetic processing.

```
sradius = input("Please enter the radius of the circle ")
radius = float(sradius)
diameter = 2 * radius
```

1.9.1. String Formatting

We have already seen that the `print` function provides a very simple way to output values from a Python program. `print` takes zero or more parameters and displays them using a single blank as the default separator. It is possible to change the separator character by setting the `sep` argument. In addition, each print ends with a newline character by default. This behavior can be changed by setting the `end` argument. These variations are shown in the following session:

```
>>> print("Hello")
Hello
>>> print("Hello","World")
Hello World
>>> print("Hello","World", sep="***")
Hello***World
>>> print("Hello","World", end="***")
Hello World***>>>
```

It is often useful to have more control over the look of your output. Fortunately, Python provides us with an alternative called **formatted strings**. A formatted string is a template in which words or spaces that will remain constant are combined with placeholders for variables that will be inserted into the string. For example, the statement

```
print(aName, "is", age, "years old.")
```

contains the words `is` and `years old`, but the name and the age will change depending on the variable values at the time of execution. Using a formatted string, we write the previous statement as

```
print("%s is %d years old." % (aName, age))
```

This simple example illustrates a new string expression. The `%` operator is a string operator called the **format operator**. The left side of the expression holds the template or format string, and the right side holds a collection of values that will be substituted into the format string. Note that the number of values in the collection on the right side corresponds with the number of `%` characters in the format string. Values are taken—in order, left to right—from the collection and inserted into the format string.

Let's look at both sides of this formatting expression in more detail. The format string may contain one or more conversion specifications. A conversion character tells the format operator what type of value is going to be inserted into that position in the string. In the example above, the `%s` specifies a string, while the `%d` specifies an integer. Other possible type specifications include `i`, `u`, `f`, `e`, `g`, `c`, or `%`. Table 9 summarizes all of the various type specifications.

Table 9: String Formatting Conversion Characters

Character	Output Format
`d`, `i`	Integer
`u`	Unsigned integer
`f`	Floating point as m.ddddd
`e`	Floating point as m.dddddde+/-xx
`E`	Floating point as m.dddddE+/-xx
`g`	Use `%e` for exponents less than -4 or greater than $+5$, otherwise use `%f`
`c`	Single character

Character	Output Format
`s`	String, or any Python data object that can be converted to a string by using the `str` function.
`%`	Insert a literal % character

In addition to the format character, you can also include a format modifier between the `%` and the format character. Format modifiers may be used to left-justify or right-justifiy the value with a specified field width. Modifiers can also be used to specify the field width along with a number of digits after the decimal point. Table 10 explains these format modifiers

Table 10: Additional formatting options

Modifier	Example	Description
number	`%20d`	Put the value in a field width of 20
`-`	`%-20d`	Put the value in a field 20 characters wide, left-justified
`+`	`%+20d`	Put the value in a field 20 characters wide, right-justified
`0`	`%020d`	Put the value in a field 20 characters wide, fill in with leading zeros.
`.`	`%20.2f`	Put the value in a field 20 characters wide with 2 characters to the right of the decimal point.
`(name)`	`%(name)d`	Get the value from the supplied dictionary using `name` as the key.

The right side of the format operator is a collection of values that will be inserted into the format string. The collection will be either a tuple or a dictionary. If the collection is a tuple, the values are inserted in order of position. That is, the first element in the tuple corresponds to the first format character in the format string. If the collection is a dictionary, the values are inserted according to their keys. In this case all format characters must use the `(name)` modifier to specify the name of the key.

```
>>> price = 24
>>> item = "banana"
>>> print("The %s costs %d cents"%(item,price))
The banana costs 24 cents
>>> print("The %+10s costs %5.2f cents"%(item,price))
The     banana costs 24.00 cents
>>> print("The %+10s costs %10.2f cents"%(item,price))
The     banana costs      24.00 cents
>>> itemdict = {"item":"banana","cost":24}
>>> print("The %(item)s costs %(cost)7.1f cents"%itemdict)
The banana costs    24.0 cents
>>>
```

In addition to format strings that use format characters and format modifiers, Python strings also include a `format` method that can be used in conjunction with a new `Formatter` class to implement complex string formatting. More about these features can be found in the Python library reference manual.

You have attempted 1 of 2 activities on this page

(GettingStartedwithData.html)

(ControlStructures.html)

Mark as completed