How does a function function?

Abstraction is a key concept in computer science and problem solving in general. It is a strategy of simplification, wherein concrete details are ‘hidden’ inside a higher-level data object. One way that programmers create an abstraction is by defining a function.

A function is a named sequence of statements that are to be executed each time the function is called. A function can be parameterized, so that it’s behavior can depend on arguments that the programmer supplies when calling the function. The arguments serve as inputs.

This exercise will reinforce your understanding of how a function is defined, how information flows into and out of a function, and how a function call is executed. The last part will also give you practice writing and testing your own function. Work on this exercise with a partner.

Part(a):

1. Open Spyder. In an IPython console. Restart the kernel (use the tool/gear icon at the top right of the console).

2. In the console, type the letter c and then hit the tab key. Tab brings up a tooltip showing all names beginning with a c that are currently defined. Scroll through the tooltip to chr and press enter/return. What is the significance of what the console displays?

3. What happens if you type a name beginning with a c that is not on this list? (Run a test to find out—e.g., enter correct)

4. Now type ch and then hit the tab key. Why does hitting the tab key complete the ch to chr?

5. The chr function is an example of a built-in function. All of the functions we have used so far are built-in functions. You can call the (built-in) help function to learn information about any object. Test this out now by entering help(chr) in the console. What do you think the information that it displays about the chr function means?

6. Run the following tests in the console to learn more about the chr function.

   \[
   \begin{array}{ll}
   \text{chr}(97) & \text{chr}(65) \\
   \text{chr}(98) & \text{chr}(9) \\
   \text{chr}(99) & \text{chr}(10) \\
   \text{chr}(97 + 25) & \text{chr}(34) \\
   \text{chr}(48) & \text{chr}(39) \\
   \text{chr}(48 + 1) & \text{chr}(1500) \\
   \text{chr}(48 + 9) & \text{chr}(1502)
   \end{array}
   \]

\[1\text{ e.g., print, input, str, int, float, abs, round, etc.}\]
7. Now download `testConversions.py` from the CTL website, open it in your current Spyder session **but do not run it yet**. The file contains two function definitions. What are the names of the two functions?

8. Because you have not yet run the file, the console has not yet executed the function definitions. Check that this is the case by typing the function names in the console and entering return.

9. Now run the program. Then type the function names in the console—what does the console return now when you enter one of the new function names and press return?

10. Now call `help(convert)`. How do you think the `help` function determines what information to display for a function, like `convert`, that you define?

   A string immediately following a function header is called a **docstring**. When defining a function, you should always provide an informative docstring so others can determine how to call the function and what the function does.

11. In the console, call the function `convert` with 100 for the first argument and 2 for the second argument. What is the result displayed? Try calling `convert` with several other arguments. What happens if you don’t use “legal” values for the arguments (i.e., if you don’t follow the instructions in the docstring)?

   **ASIDE:** If you are curious about the algorithm used for `convert`, I recommend the series of Kahn Academy videos at the last link in the Artifacts section of this week’s website. If you are a CSE major, you will learn these algorithms in CSE 260.
Part (b): Press the Visualization link in the Artifacts section of the website for this week. Before stepping through the visualization, observe that the program consists of the two function definitions from testConversions.py and a call, test(), to the test driver function.

With your partner, take turns predicting what will happen at each step and then checking your understanding by advancing to the next step. Step through the visualization in this manner.

Answer the following questions based on your observations during the visualization. Remember, the goal is not to fill answers into a worksheet—rather, it is to understand how the console executes the function calls. So put up your pink tent and ask about anything you don’t understand or to check that your understanding is correct!

Q1: What side effect does executing a function definition produce?

Q2: What names are in the following namespaces:

- Global namespace
- Local namespace of convert
- Local namespace of test

Q3: What are the n and b in the header and the body of convert called?______________

What are the values of n and b
- The first time that convert is called? n = _______ b = _______
- The second time that convert is called? n = _______ b = _______

Q4: When control (the red arrow) is at the assignment statement n = n // b in the body of convert (e.g., at step 14), how many namespaces contain the name n? How does the console know which value for n to use?
Part (c): In this part, you will add another function definition to `testConversions.py`. The function will be named `print_conversion_table`. It will be called with two arguments: the first will be an integer base value, $b$, and must be between 2 and 36 (inclusive); the second will be a positive integer, $n$. When called, the function will print a table showing the base-$b$ conversion for all decimal numbers from 0 through $n$. It will not return any value. It will call `convert` to perform the base conversions.

For example: calling our `print_conversion_table(2, 16)` prints

<table>
<thead>
<tr>
<th>$n$</th>
<th>base-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
</tr>
</tbody>
</table>

Suggestion: Don’t worry about printing a nicely formatted table to start off with. Write a first version that just prints each decimal number and its conversion. If there is time remaining once you have that working, add logic to format the table nicely. (You will need to decide how wide to make the columns based on the width of the largest value to be displayed in a column. We used the `rjust` and `format` methods to line up values within columns.)