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 formerly 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.

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 several parts will also give you practice with writing your own functions. Work on this exercise with a partner.

**Part(a):** [tooltips, shell completion, built-in functions, *help*, *chr*, and *ord*]

1. Open Spyder. In an *IPython console*, restart the kernel. In the rest of this exercise, we refer to this console more simply as the *shell*.

2. In the shell, type the letter *c* and then hit the *tab* key. This brings up a *tooltip* showing all names beginning with a *c* that are currently defined (and some other things). What happens if you type a name beginning with a *c* that is not on this list? (Run a test to find out.)

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

4. The *chr* function is an example of a *built-in function*. All of the functions we have used so far (e.g., *print*, *input*, *str*, etc.) are built-in functions. There is also a built-in *help* function that you can call to learn information about any object. Test it out now by entering *help(chr)* in the shell. What can you conclude about how the function *chr* is called and what it returns? Call *chr* in the shell several times with different (legal) arguments to test your understanding.

5. Enter *help(ord)* in the shell. What can you conclude about how the built-in function *ord* is called and what it returns? Call *ord* in the shell several times with different (legal) arguments. What is the relationship between *chr* and *ord*? (Hint: You may want to compare the results of your tests to the table at [http://www.asciitable.com/](http://www.asciitable.com/))
Part (b): [Visualizing execution of calls to functions] Download `testConversions.py` into a new folder created for today. Open the downloaded file in Spyder, **but do not run it yet.** Instead, create a Python Tutor visualization of it as follows:

1. In a web browser, go to [http://pythontutor.com](http://pythontutor.com) -> “Start using Python Tutor now.”
2. Above the edit box, select “Python 3.3” if it is not already selected.
3. Below the edit box, in the middle menu, select “render all objects on the heap”.
4. Paste the contents of `testConversions.py` into the Python Tutor edit box. (Select them in the Spyder edit pane and copy (`⌘c`)/paste (`⌘v`) them into the Python Tutor edit box.)
5. Press the “Visualize Execution” button in the bottom left of the edit window.

With a partner: take turns predicting how executing the next step will affect the visualization. Then check your understanding by pressing the “forward button.” Continue stepping through the program until you both understand how the functions are called and executed.

Answer the following questions based on your observations during the visualization. Remember, the goal is not to fill correct answers into a worksheet—rather, it is to understand how to find the answer. **So, if you don’t understand the reason for an answer, ask!**

Q1: What names are in the following namespaces:

- The global namespace

- The local namespace of `convert`

- The local namespace of `revert`

Q2: What values are assigned to the functions’ parameters:

- The first time `convert` is called (i.e., when step 8 is executed)?
  - `n` = ________  `b` = ________

- The first time `revert` is called (i.e., when step 31 is executed)?
  - `s` = ________  `b` = ________

- The second time `convert` is called (i.e., when step 50 is executed)?
  - `n` = ________  `b` = ________

- The second time `revert` is called (i.e., when step 68 is executed)?
  - `s` = ________  `b` = ________

Q3: When control (the red arrow) is at the assignment statement `n = n // b` in the body of `convert` (e.g., at step 15), how many namespaces contain the name `n`? Which `n` is used?
**Part (c):** [Executing function definitions, Doc strings] Return to your Sypder session.

1. Before running the program `testConversions.py`:
   a. In the shell, type `con` and then press the `tab` key. What can you conclude about the name `convert` from this experiment?
   b. Run a similar experiment for `revert` – e.g., type `rev` and then press the `tab` key. What can you conclude about the name `revert`?

2. Now run `testConversions.py`. Then, in the shell, type `con` a second time and then hit the `tab` key. Why does it behave differently now than in step 1a? Run a similar experiment for `revert`.

3. Next enter an open parenthesis (“(“). How does Python know what to display in the tooltip?

4. Now enter `help(convert)`. Notice that the help function displays the string that immediately follows the function header. This string is called a docstring. The programmer who writes a function should always provide an informative docstring so others can know the purpose of the function and how to call it.

5. In the shell, 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 a few other arguments. What happens if you don’t use “legal” values for the arguments (i.e., you don’t follow the instructions in the docstring)?

6. In the shell, assign variables `a` and `b` some positive integer values; make `a` as large or small as you want, but assign `b` a value between 2 and 36.
   a. Call `convert` with `a` for the first argument and `b` for the second argument.
   b. Call `convert` with `b` for both the first and second arguments.
   c. Call `convert` with `3*a` for the first argument and `b` for the second argument.
   Be sure you understand why these calls are all producing different results and what results they are producing.

7. In the shell, type `rev` and then enter `tab`. Complete it to `revert`. Add an open parenthesis to learn how the function is to be called. Then call it several times using different base-b strings and values for `b`. For example, you can call it with some of the strings returned by your previous calls to `convert` and the base used for the conversion.

8. Finally, in the Python shell, type `t` and enter `tab`, then add parentheses with no argument in between, and enter `return`. Explain the output that is produced.

**ASIDE:** If you are curious about the algorithm used for `convert` and `revert`, 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 (d): In this part, you will add three other function definitions to `testConversions.py`. But before adding the definitions, delete or comment out the call to function `test`.

The first two functions will be named `get_conversion_limit` and `get_conversion_base`. They will be used to gather inputs from a user. Neither will take any arguments. The function `get_conversion_limit` will request a user to enter a conversion limit, which must be a positive integer. If the user enters an illegal string, the function will make the request again. This behavior will repeat until the user enters a legal string. The function will return the `int` value of the string entered.

The function `get_conversion_base` will behave in a similar manner. However, it will request a user to enter a base, which must be between 2 and 36, inclusive.

To clarify, the following examples illustrate the behavior of our functions.

```
In [6]: get_conversion_limit()
Conversion limit must be a positive int.
Enter the conversion limit: 0
Conversion limit must be a positive int.
Enter the conversion limit: -20
Conversion limit must be a positive int.
Enter the conversion limit: 1.0
Conversion limit must be a positive int.

Enter the conversion limit: 1
Out[6]: 1

In [7]: get_conversion_base()
Base must be at least 2 and no more than 36.
Enter the base for the conversion: ?
Base must be at least 2 and no more than 36.

Enter the base for the conversion: 1
Base must be at least 2 and no more than 36.

Enter the base for the conversion: 2
Out[7]: 2
```

Hint: You will find the string `isdigit` method useful for checking if a string denotes an `int` value. If you are not already familiar with this method, run a few experiments in the shell (e.g, `'535'.isdigit()` and `'hi3'.isdigit()). For more information about it, enter `help("".isdigit())`. 
The third function will be named `print_conversion_table`. It will be called with two arguments: the first will be an integer base value, `base`, and must be between 2 and 36 (inclusive); the second will be a positive integer, `limit`. When called, the function will print a table showing the Base-`base` conversion for all decimal numbers from 0 through `limit` (inclusive). It will not return any value. It will call `convert` to perform the base conversions.

For example:

```
In [15]: print_conversion_table(2, 16)

<table>
<thead>
<tr>
<th>#</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>
```

**But do not worry about printing a nicely formatted table yet.** Write a first version that just prints each decimal number and its conversion to a line. Go on to the next part.

**Part (e):** [main function] Next add a `main` function which uses the functions defined in part (d) to request a conversion limit and a conversion base from a user, and then prints a conversion table using the base and limit provided by the user. The main function should not take any arguments. Add a call to `main` at the bottom of the program.

**Part (f):** [formatting] Once you have this 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 both the `rjust` and `format` methods to line up values within columns.)
Part (g): [Extra formatting for experts: multiple “columns”] Modify function `print_conversion_table` so that it takes an optional third argument, which defaults to 1 and signifies the number of columns, `col`, to use in formatting the table, where a decimal number and its base conversion is viewed as a column (containing 2 subcolumns). The numbers and their conversions are to be displayed lined up in `col` columns in order from left to right, wrapping to the next row when needed.

For example:

```
In [21]: print_conversion_table(16, 45, 3)
```

<table>
<thead>
<tr>
<th>#</th>
<th>Base-16</th>
<th></th>
<th>#</th>
<th>Base-16</th>
<th></th>
<th>#</th>
<th>Base-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>4</td>
<td>4</td>
<td></td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td>7</td>
<td>7</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
<td>10</td>
<td>A</td>
<td></td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td></td>
<td>13</td>
<td>D</td>
<td></td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td></td>
<td>16</td>
<td>10</td>
<td></td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td></td>
<td>19</td>
<td>13</td>
<td></td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td></td>
<td>22</td>
<td>16</td>
<td></td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td></td>
<td>25</td>
<td>19</td>
<td></td>
<td>26</td>
<td>1A</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td></td>
<td>28</td>
<td>1C</td>
<td></td>
<td>29</td>
<td>1D</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td></td>
<td>31</td>
<td>1F</td>
<td></td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td></td>
<td>34</td>
<td>22</td>
<td></td>
<td>35</td>
<td>23</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td></td>
<td>37</td>
<td>25</td>
<td></td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td></td>
<td>40</td>
<td>28</td>
<td></td>
<td>41</td>
<td>29</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td></td>
<td>43</td>
<td>2B</td>
<td></td>
<td>44</td>
<td>2C</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the example in part (d) is also accurate since the default value for `col` is 1.

Finally, modify `main` so that it prints the conversion table in 1 column if the user enters a limit that is 6 or fewer; in 2 columns if the limit is between 7 and 19, inclusive; 3 columns if the limit is between 20 and 44, inclusive; and 4 columns if the limit it 45 or more.