Discussion section exercises for November 12-13: Optional arguments

Most of what we do in this course involves generic computer science ideas: Branching (if), looping (for and while), recursion, algorithms for manipulating, searching and sorting lists, hash tables...these are fundamental building blocks that you will find in any programming language, and any discussion of computer algorithms.

A few things, though, are peculiar to Python. This discussion section is devoted to the way that Python allows you to write functions that accept a variable number of arguments, and functions with ‘keyword’ arguments. Knowing how this works won’t translate to other programming languages, but it will transform you into a Python master.

1. Python fixes the number of arguments of some functions, but allows others to be called with an arbitrary number of arguments. For example,

   ```python
   import math
   math.sqrt(4,3)
   ```

   will lead to an error (try it to see what the error message says), but

   ```
   max(2,3)
   max(9-2,18,43,26)
   ```

   are both ok. Let’s see how to write functions that allow this behavior.

   (a) Run the function f1 in the code provided, typing, successively, f1(3), f1(5,7,-2), f1() at the prompt in the Python shell. You should see that the presence of the parameter preceded by a * causes the sequence of arguments to be gathered into a tuple.

   (b) Use this behavior to write a function `average` that takes any positive number of arguments and returns their average. For example, you should get the following results:

     ```
     >>> average(2)
     2.0
     >>> average(2,3,4,5)
     3.5
     >>> average()
     average requires at least one argument
     ```
2. Look at the function f2 in the code provided.
(a) You can probably figure out what this code is doing, but in any case
test it by typing

\[ f2(435,10) \]

and test it again by typing

\[ f2(10,435) \]

In these examples, the arguments 10 and 435 are ‘positional arguments’. As you
know by now, the order here matters, and to use functions like this effectively, you
need to remember the order in which the arguments go.

(b) But there is another trick you can try. Type

\[ f2(base=10,number=435) \]

Now we have used `base` and `number` as ‘keyword arguments’. The names of the
parameters in the function are not completely arbitrary, and calling the function in
this way binds each argument to the parameter with the same name, instead of by
the order. (I learn something new every time I teach this course. I did not know you
could do this!)

(c) Here is a variant that may look familiar, because we used something like it in an
earlier assignment. Revise the code for `f2` so that the first line reads

```python
def f2(number,base=10)
```

Here we give a default value to the parameter labeled `base`. Now try out various
combinations such as

\[ f2(437), f2(437,2), f2(base=2,number=437), f2(base=2,437). \]
(They’re not all legal!)

3. The function f3 in the provided code combines the approaches of the last two
problems.

(a) First try typing

\[ f3(2,3,4) \]

and then

\[ f3(2,3,4,option=6). \]
(b) Change the code in the function definition so that the first line reads

```python
def f3(*u, option=2)
```

and try the two lines above again.

(c) Now write a useful function that uses this behavior. Rewrite your `average` function (call the new version `average2`) so that it includes an additional keyword argument `Integer`, which by default has the value `False`. When this argument is set to `False`, your function should compute the average as it would normally. When the argument is `True`, it will round the result to the nearest integer (use the built-in `round` function). So, for example

```python
>>> average2(2, 3, 4, 5)
3.5
>>> average2(2, 3, 4, 5, Integer=True)
3
```

4. The function `f4` shows a fancier approach to keyword arguments, which allows us to use a large number of such arguments optionally without necessarily providing default values.

(a) Type

```python
f4(2, 3, 4, 5, weights=[0.1, 0.6, 0.2, 0.1], Integer=True)
```

to see what this does.

(b) When you introduce an argument with `**`, it gathers all the keywords in the function call together with their values into a dictionary. Use this behavior to rewrite the `average` function (call it `average3`) one more time, with keywords `weights` and `Integer`. If the `weights` keyword is not present, the function will work as in the preceding problem. If it is present, then the function will return the weighted sum of the initial arguments. (Don't worry about checking some things that really should be checked here---such as that the length of the list of weights is the same as the number of arguments, that the weights add up to 1, and that no keywords other than these two are used.)

You should get the following results:

```python
>>> average3(2, 3, 4, weights=[0.3, 0.3, 0.4])
3.1
>>> average3(2, 3, 4, weights=[0.3, 0.3, 0.4], Integer=True)
3
>>> average3(2, 3, 4)
3.0
```