Hw 1, Part 2 (Lab): Functioning smoothly!

Using built-in functions

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/Lab1B on 3/20/2017

First, try out some of Python's many *built-in* functions. These will *not* go into your `hw1pr2.py` file:

```
In [1]: list(range(0,100))
Out[1]: [0,1,2,...,99]
```

`range` returns an iterator of integers. `list` turns this into a list of integers.
Note that when you use `range`, as with almost everything in python,
the **right endpoint is omitted!**

```
In [2]: sum(range(3,11))
Out[2]: 52
```

`sum` sums a list of numbers, and `range` creates a list of integers.

```
In [3]: sum([40,2])
Out[3]: 42
```

a roundabout way of adding `40+2`

```
In [4]: help(sum)
(an explanation of sum)
```

`help` is a good thing—ask for it by name!
Depending on your computer, you may need to hit `q` in order to leave the *help* interface...

```
In [5]: import math
```

You do need to `import math` first...or else you'll get an error!

```
In [6]: dir( math )
```

All of the functions in the `math` library...

```
In [7]: dir( __builtins__ ) (a huge list of the math library's and the built-
in functions, respectively)
```

This `dir` command is also useful - it lists everything from a particular library (or *module*, as it's also called). The special `__builtins__` module holds all of the built-in functions!

To type `__builtins__`, there are two *underscores* both before and after the lowercase letters: the underscore, _, is shift-hyphen on most keyboards (between the right parenthesis and the plus sign).

If you look carefully in the big list of stuff returned from `dir(_ _ builtins _ _)`, you will find the `sum` function you used above. You will **not**, however, find some other important functions, for example, `sin` or `cos` or `sqrt`. All of these functions (and many more) are in the `math` module. There are many modules (libraries) of functions available for you to use as part of Anaconda Python, along with even more available for download beyond that foundation.

Importing other code (or "modules")

To access functions that are not built-in by default, you need to load them from their modules or libraries. (We use those terms interchangeably.) Try out these examples to get familiar with how to access Python's many libraries:

(1) You can import a module, i.e., a library, and then access its functions with that module name:

```
In [1]: import math
(no response from Python)

In [2]: math.sqrt(9)
Out[2]: 3.0
```

Note that `sqrt` returns a `float` even if its input is an `int`.

```
In [3]: math.cos(3.14159)
Out[3]: -0.999...
```

Note that `cos` et al. take radians as input. Also, 3.14159 is less than `math.pi`.

(2) Tired of typing `math.` in front of things? You can avoid this with

```
In [1]: from math import *
(no response from Python)
```

The asterisk `*` here means "everything." This will bring all of the functions and constants from the math module into your current python environment, and you can use them without prefacing them by `math.`

```
In [2]: cos(pi)
Out[2]: -1.0
```

This would have had to be `math.cos(math.pi)` before the new "`from math import *`" import statement.

Note: It's not *always* the best idea to do this...especially when you're using lots of libraries - and some may share the same function name.

Create a new file named `hw1pr2.py`

Next, you'll create a few functions *of your own* in a new file, `hw1pr2.py`.

So, use your text-editor to create a new file named `hw1pr2.py`.

(*Note*: CS 5's lectures will cover functions in detail at the next meeting. Here, in anticipation, you'll create a few functions just to get started... .)

Functions are the fundamental building blocks of computation. What distinguishes Python from other computing environments is the ease and power of creating your own functions!

Start by pasting the following comments and a definition of a function named `dbl`:

```
# CS5 Gold, Lab1 part 2
# Filename: hw1pr2.py
# Name:
# Problem description: First few functions!


def dbl(x):
    """  output: dbl returns twice its input
         input x: a number (int or float)
         Spam is great, and dbl("spam") is better!
    """
    return 2*x
```

Run your file

When you run this file in the usual way:

```
run hw1pr2.py
```

you won't see anything... . However, your newly defined function, `dbl` is now available.

From here, *try using* the newly defined `dbl` function:

```
In [1]: dbl(21)
Out[1]: 42

In [2]: dbl('wow! ')
Out[2]: 'wow! wow!'
```

Signature and Docstring

The first line of a Python function is called its ***signature***. The function signature includes the keyword `def`, the name of the function, and a parenthesized list of inputs to the function.

Docstring Directly underneath the signature is a string inside triple quotes `"""` - this is called the ***docstring***, short for "documentation string." CS5 asks you to include a docstring in all of your functions (even simple ones such as these, in order to feed the habit).

A docstring should describe what the function inputs and outputs. As you see above, it may include other important information, too. Docstrings are how *your* functions become part of Python's built-in help system.

To see this, type

```
In [2]: help(dbl)
```

and you will see that Python provides the docstring as the help! The language's help system is docstrings! This self-documenting feature in Python is especially important for making your functions understandable, both to others and to yourself.

Important Warning: the first set of triple quotes of a docstring needs to be indented underneath the function definition `def` line, at the same level of indentation as the rest of block of code that defines the function.

Writing your own functions...

Let's go!

For each of these functions, be sure to include a docstring that describes **what your function does** and **what its inputs are** for each function. See the `tpl` example, below, for a reasonable starting point and guide:

Example problem: Write the function `tpl(x)`, which takes in a numeric input and outputs three times that input.

Answer to example problem: Copy the following solution (after a few blank lines to leave space and help readability) into your `hw1pr2.py` file:

```
def tpl(x):
    """ output: tpl returns thrice its input
         input x: a number (int or float)
    """
    return 3*x
```

The five functions to write...

1. Write `sq(x)`, which takes in a number named `x` as input.
   Then, `sq` should output the square of its input.
   Note that this is the *square*, not the square root. (The square is `x` times itself... .)

2. `interp(low,hi,fraction)` takes in three numbers, `low`, `hi`, and `fraction`, and should return the floating-point value that is `fraction` of the way between `low` and `hi`.

   What!?

   That is to say, if `fraction` is zero, `low` will be returned. If `fraction` is one, `hi` will be returned, and values of `fraction` between `0` and `1` will lead to results between `low` and `hi`. (In fact, values of `fraction` can go below `0`, yielding outputs less than `low`, and they can go above `1`, producing outputs greater than `high`. Purists would call this extrapolation, rather than interpolation, however.)

From the above description, it might be tempting to divide this function into several cases and use `if`, `elif`, and the like. Yet, this function can be written using ***no*** conditional (`if/elif/else`) constructions at all! Try it *without* using `if`!

As noted, your function should also work if `fraction` is less than zero or greater than one. In this case, it will be linearly extrapolating, rather than interpolating. We'll stick with the name `interp` anyway.

Here are examples that will help clarify how `interp` works:

```
In [1]: interp(1.0, 9.0, 0.25)      # a quarter (.25) of the way from
1.0 to 9.0
Out[1]: 3.0

In [2]: interp(1.0, 3.0, 0.25)      # a quarter of the way from 1.0 to
3.0
Out[2]: 1.5

In [3]: interp(2, 12, 0.22)         # 22% of the way from 2 to 12
Out[3]: 4.2
```

Hint: If you're unsure of where to begin on this problem, look at the first example above. In it

```
 low is 1.0
 hi is 9.0
 fraction is 0.25
```

See if you can determine how to combine those three values to yield the correct output of `3.0` . (Consider starting with `(hi - low)`)

Here are two more examples to try:

```
In [1]: interp(24, 42, 0)           # 0% of the way from 24 to 42
Out[1]: 24.0

In [2]: interp(102, 117, -4.0)        # -400% of the way from 102 to
117 (whoa!)
Out[2]: 42.0
```

The next several functions involve strings of characters. Write each one in your `hw1pr2.py` file. After you write each function, be sure to test it! Also, be sure to include a docstring for each function that tells (very briefly) what it does.

3. Write a function `checkends(s)`, which takes in a string `s` and returns `True` if the first character in `s` is the same as the last character in `s`. It returns `False` otherwise. The `checkends` function does not have to work on the empty string (the string `''`).

   There is a hint below, but read through the examples first.

   These examples will help explain `checkends` - read them over now and be sure to try them once you have a first draft of your function. Notice that the final, fourth example below is the *string of one space character*, which is different from the empty string, which contains no characters:

   ```
   In [1]: checkends('no match')
   Out[1]: False

   In [2]: checkends('hah! a match')
   Out[2]: True

   In [3]: checkends('q')
   Out[3]: True

   In [4]: checkends(' ')
   Out[4]: True
   ```

   Make sure to check that this last example (the string of a single space) works for your `checkends` function. The empty string does not need to work.

   Hint: For this function you could use an `if` and `else` construction... here is a start:

   ```
       if s[0] == ______ :
           return True
       else:
           return False
   ```
   You might find a solution that doesn't use the `if` and `else` at all—this is fine, too. Notice that the *last* character is missing above—you'll need to fill that in!

   Warning! Your function ***should not return strings***! Rather, it should return a *boolean value*, either `True` or `False`, without any quotes around them. These `True` and `False` are keywords recognized by Python as representing one bit of information.

You'll see these booleans turn a different color, purple in Sublime's default color scheme, indicating that Python does recognize them as `bool` values. (If you'd accidentally made them strings, they'd be quoted, and they'd be yellow in Sublime.) In sum, booleans and strings are different. For `True` and `False` you would almost always want the unquoted boolean values.

4. Write a function `flipside(s)`, which takes in a string `s` and returns a string whose first half is `s`'s second half and whose second half is `s`'s first half. If `len(s)` (the length of `s`) is odd, the first half of the input string should have one fewer character than the second half. (Accordingly, the second half of the output string will be one shorter than the first half in these cases.) There's also a hint after the examples below.

   Here you may want to use the built-in function `len(s)`, which returns the length of the input string, `s`.

   Examples:

```
In [1]: flipside('homework')
Out[1]: workhome

In [2]: flipside('carpets')
Out[2]: petscar
```

   Hint: This function is simpler if you create a variable equal to `len(s)//2` on the first line, e.g.,

```
def flipside( s ):
    """ put your docstring here
    """
    x = len(s)//2
    return   ____________
```

   where the return statement has been left up to you... it will use the variable `x` *twice* , which is why it's nice to give it a name, rather than type and re-type it!

5. Write `convertFromSeconds(s)`, which takes in a nonnegative **integer** number of seconds `s` and returns a list (we'll call it `L`) of four nonnegative integers that represents that number of seconds in more conventional units of time, such that:
   - the initial element represents a number of days
   - the next element represents a number of hours
   - the next element represents a number of minutes
   - the final element represents a number of seconds

   You should be sure that

   - 0 ≤ seconds < 60
   - 0 ≤ minutes < 60
   - 0 ≤ hours < 24

   There are no limits on the number of days.
   For instance,

```
In [1]: convertFromSeconds(610)
Out[1]: [0, 0, 10, 10]

In [2]: convertFromSeconds(100000)
Out[2]: [1, 3, 46, 40]
```

   How to do this? Feel free to copy-and-paste this starter code that uses four variables:

```
def convertFromSeconds( s ):
    days = s // (24*60*60)  # # of days
    s = s % (24*60*60)     # the leftover
    hours =
    minutes =
    seconds =
    return [days, hours, minutes, seconds]
```

   The idea here is that, when those four variables are all correctly set, you can return them all in a list, which is the final line:

```
    return [days, hours, minutes, seconds]
```

   For instance, the line that sets `days` could be

```
    days = s // (24*60*60)
```

   What would be other lines be?

   It's possible to do this without changing the variable `s` at all. However, as the above starting code suggests, it's also possible to *alter* `s` as you go. Try this latter approach, just to get the hang of this powerful strategy.

Submitting your file

Congratulations! You have completed the second problem of Lab 1. You should submit your `hw1pr2.py` file in the usual way. Even if you have not completed the Lab's problems completely, submit what you have and you will receive full credit for both of the problems (as long as you've come to lab and given a full effort).

You now have a choice: you could

1. Continue working on the rest of HW1
2. Leave!

Either one is OK, but when you do leave, be sure you've signed the attendance list for the lab!