Homework 3

Functions

1. The Laplace Distribution is a continuous probability distribution with a shape that's determined by a center parameter μ and a spread parameter b. Write a function that returns the probability density of the Laplace distribution evaluated at some real number value `x` and call the function `lap_pdf`. Allow the user to pass two optional arguments specifying the parameters of the distribution that are otherwise set to default values of 0 and 1 respectively. Test out your function at a view values of `x` and a couple parameter values to test if it's working the way you expect it to.

2. Consider the following function:

```
x = 1
def f(y):
    global x
    x = 5
    return x + y
```

    1. Does this function have access to the value of `x` in the global namespace even though you haven't passed it in as an argument? Write an analagous function in R and compare the behavior.
    2. Add a line to the function before the `return` where you reassign `x` to have a new value. Does running the function change the value that `x` takes in the global namespace? Compare this behavior to R.
    3. Add one more line above the previous one to declare that `x` is a global variable (as opposed to a local variable): `global x`. How does this change the behavior of the function? Compare it to a similar modification in R where you instead reassign `x` with the super-assignment operator: `<<-`.

3. In class I gave an ill-conceived poll that asked you to, "Write a function that takes a given string, and outputs it as a single string repeated `n` times, each one separated by a `.`" We saw that a naive implementation does not work:

```
def dotted_print(x, n):
    return print(x * n, sep = ".")

dotted_print("hello", 3)
```

```
## hellohellohello
```

The reason is that `x * n` is evaluated first, which smushes the repeated strings together to form a single scalar string. With only one string there (albeit one with repetitions inside it), `print()` has nothing to glue together with a `..`

You can fix this behavior by being sure that each repeated string remains a separate element before getting passed to `print()` and that print recognizes each of those elements to be separate. You can get there by incorporating two realizations:

- As we've seen, you can use `*` and `+` operators with string types to repeat and concatenate them. They can also be used on list types with a similar functionality. The list type is one way to preserve the separation between the repeated strings.
- The `*` character can also be used before a variable passed to a function (this character keeps very busy in Python!). This can be read to mean, "expand this as a sequence" (see p. 43 in *Whirlwind*). If you look in `?print`, this allows you to pass a compound data structure not as a single `value`, but as multiple values that take advantage of the `...`.

Use these ideas to fix `dotted_print()`.

Methods

1. Consider the following list of strings:

```
l = ["my", "it's", "smokey", "out"]
```

    1. Query two attributes of this string that are of interest to you.
    2. Use the `.append()` list method to add the string `"today"` to this list.
    3. Use list comprehension (see Lab 2) and string methods to capitalize each of the words in this list.
    4. Construct one addition list comprehension that uses a string method of your choice and asserts a logical condition for the operation using the `if` keyword.

2. One of the most common places that you'll find object-oriented programming in R is in print methods. Any time you type the name of a variable at the console, it often doesn't actually print out the value of that object, but rather calls on it's appropriate print method.

    1. Create a linear model object using `lm()` (you're welcome to use the `mtcars` example from the slides) and call it `m1`. What is the class of `m1`? What is its type (use `typeof()`)? What do you get when you print it to the console by either just typing the of the object or using `print()`? Compare that with what you get if you print the object using `print.default()`.
    2. Run the `summary()` of the model object and save it as `s1`. Answer the same questions as above for `s1`.
    3. Create a ggplot object and save it as `p1`. Answer the same questions as above for `p1` and also look at the help file for `print.ggplot()`.