Lab 11: Scatter Plots, Recursion, and Shortcuts

In this lab, you'll learn how to create scatter plots, practice writing recursive code, and learn two techniques (default dictionaries and list comprehensions) to shorten your code. Start by creating a notebook for this lab.

Scatter

We'll eventually learn much more about scatter plots, but for now, just copy/paste these two cells to your notebook:

Cell 1:

```
%matplotlib inline
```

Cell 2:

```
import pandas as pd

def scatter(x, y, xlabel="please label me!", ylabel="please label me!"):
    df = pd.DataFrame({"x":x, "y":y})
    ax = df.plot.scatter(x="x", y="y", color="black", fontsize=16, xlim=0,
ylim=0)
    ax.set_xlabel(xlabel, fontsize=16)
    ax.set_ylabel(ylabel, fontsize=16)
    ax.get_xaxis().get_major_formatter().set_scientific(False)
    ax.get_yaxis().get_major_formatter().set_scientific(False)
```

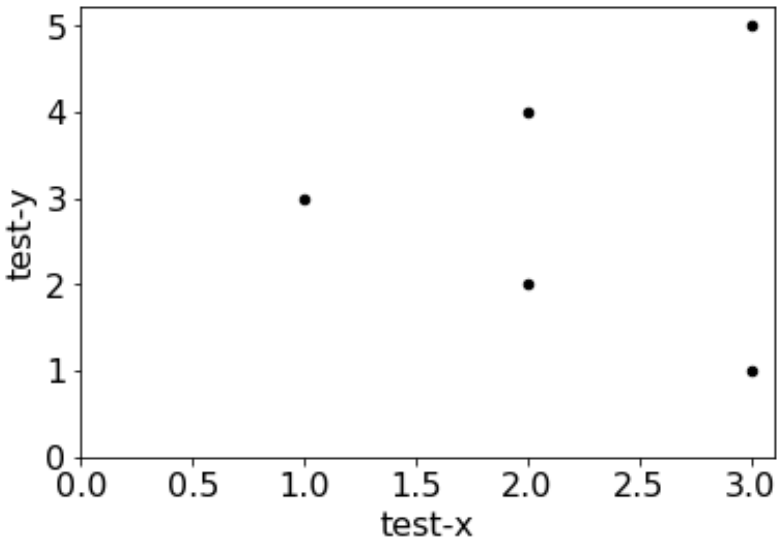
Let's try it (paste/run the following):

```
scatter([0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0, 8.0, 8.0,
8.0],
        [4.1, 4.3, 4.2, 5.8, 6.0, 6.4, 8.4, 8.0, 7.9, 8.3, 8.1, 8.2, 1.9, 2.1,
1.0])
```

We need some axis labels; try setting them:

```
scatter([0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0, 8.0, 8.0,
8.0],
        [4.1, 4.3, 4.2, 5.8, 6.0, 6.4, 8.4, 8.0, 7.9, 8.3, 8.1, 8.2, 1.9, 2.1,
1.0],
        xlabel="ice cream scoops eaten", ylabel="happiness (out of 10)")
```

To make sure you understand how to use `scatter`, try to call it to make a plot that looks like the following:



Recursion

As we learned in class, a function is recursive when it calls itself, directly or indirectly. We'll give you some practice here completing some recursive code.

Problem 1: Factorial

Yes, we did this in class, but see if you can complete it from memory (or by deducing the answer) before referring back to your notes:

```
def fact(n):
    if ????:
        return 1
    return n * ????
```

Here's a hint (the following is math, not Python):

```
1! = 1
2! = 1 * 2 = 2
3! = 1 * 2 * 3 = 6
4! = 1 * 2 * 3 * 4 = 24
```

When you're done, test your code with a few different inputs. To better understand what's happening, here are a couple things you could do:

- run your code in Python Tutor
- add a print to the beginning of your function, like this: `print("fact(" + str(n) + ") was called")`

Problem 2: String Reversal

We want `reverse` to reverse the letters in a string. So, for example, `reverse("NACIREMA")` should return "AMERICAN".

```
def reverse(s):
    if len(s) <= 1:
        return ????
    return reverse(????) + ????
```

Hints:

- using indexing and slicing on s
- notice that "ABCD" reversed is "BCD" reversed (i.e., "DCB"), concatenated with "A"

Try your function with a few strings.

Problem 3: List Reversal

Write a function that reverses a list. So, for example, `list_rev([1,2,3])` should return `[3,2,1]`. Both lists and strings are sequences, so the code should be very similar to your string reversal function. In fact, we recommend you start by copying that code, calling it with a list, then identifying the one reason the your previous function doesn't work for lists as well as strings.

Hint:

- `[1,2,3]+4` is invalid, but `[1,2,3]+[4]` performs list concatenation.

Challenge: can you devise a single function that works for all types of sequences, including strings, lists, and tuples?

Problem 4: Dictionary Printer

Complete the following function so that it prints nested dictionaries in an easy-to-read way:

```
def dprint(d, indent=0):
    print("Dictionary:")
    for k in d:
        v = d[????]
        print(" " * indent, end="")
        print(k + " => ", end="")
        if type(????) == dict:
            dprint(v, ????)
        else:
            print(v)
```

A call to `dprint({"A": 1, "B": {"C": 2, "D": 3, "E": {"F": 4}}, "G": 5})` should print the following:

```
Dictionary:
A => 1
B => Dictionary:
  C => 2
  D => 3
  E => Dictionary:
    F => 4
G => 5
```

Shortcuts For Filling Lists and Dictionaries

For those of you who have become comfortable with creating lists and dictionaries, now is a good time to learn some shortcuts. You won't be expected to know these, but they may save you some time.

If you still feel less than comfortable inserting a bunch of values into lists and dictionaries, you may want to skip this section and keep practicing the basics.

List Comprehensions

Consider this code, which creates a new list by adding 1 to every number in a prior list:

```
nums = [500, 100, 200, 300, 400]
new_nums = []
for x in nums:
    new_nums.append(x+1)
new_nums
```

This pattern of building a list inside a for loop is so common that Python provides a shorthand to do the same with a single line of code. The following example is equivalent to the above example:

```
nums = [500, 100, 200, 300, 400]
new_nums = [x+1 for x in nums]
new_nums
```

This shorthand is called a *list comprehension*, and it looks like a for loop inside a list. Things to note that are different that regular for loops:

- there's an expression BEFORE the `for`
- there's no colon after the loop

Try completing the following code to create a listing like this: `[1000, 200, 400, 600, 800]`.

```
nums = [500, 100, 200, 300, 400]
doubles = [???? for orig_num in ????]
doubles
```

It works for string too. Try completing this:

```
words = ["Apple", "banana", "ORANGE"]
upper_words = [word.upper() for ???? in ????]
upper_words
```

You're trying to get this: `["APPLE", "BANANA", "ORANGE"]`.

Default Dictionaries

Try running this:

```
fruit_counts = {}
fruit_counts["apple"] = 10    # line 1: works
print(fruit_counts["apple"])  # line 2: works
print(fruit_counts["banana"]) # line 3: fails
```

Notice that when inserting values, we can use new keys (line 1), and they are automatically added for us. But when looking up values, it fails if the key hasn't been seen before (line 3).

There's an alternative to `dict` called `defaultdict` that can provide a default value instead of failing in situations such as line 3 above.

Let's try it:

```
from collections import defaultdict

fruit_counts = defaultdict(int)
fruit_counts["apple"] = 10    # line 1: works
print(fruit_counts["apple"])  # line 2: works
print(fruit_counts["banana"]) # line 3: works
```

Notice two things:

- we need to import defaultdict before we can use it
- when we create the defaultdict object, we need to tell it what type of things it should contain so that it can provide reasonable defaults (e.g., 0 for int)

Lists are also interesting default values. Consider these two examples:

With regular dictionary:

```
rows = [
     ("alice", 5),
     ("bob", 6),
     ("alice", 7),
     ("bob", 8),
]

scores = dict()
for row in rows:
    if not row[0] in scores:
        scores[row[0]] = [row[1]]
    else:
        scores[row[0]].append(row[1])

scores
```

With default dictionary:

```
from collections import defaultdict

rows = [
     ("alice", 5),
     ("bob", 6),
     ("alice", 7),
     ("bob", 8),
```

```
]

scores = defaultdict(list)
for row in rows:
    scores[row[0]].append(row[1]) # creates a new list if necessary

dict(scores)
```

Try completing the following to count letter frequencies:

```
from collections import defaultdict

letter_counts = defaultdict(????)

for letter in "banana":
    letter_counts[????] += ????

dict(letter_counts)
```