Lecture 17

The "Downside" of Scoping; Making a Modular Program; Lists/Mutables as Arguments

1. The "Downside" of Scoping

Last time we talked about scoping: this refers to how variables defined in functions (local variables) are only accessible within those functions, not outside. Scoping helps avoid name conflicts; if you accidentally create two variables with the same name in different functions, Python will be able to keep them straight.

Scoping does have one downside: a function can't refer to variables that are defined outside of itself. (Technically, you can use global variables if they are *only* read, and *never* assigned values in the function -- but even then, it is strongly frowned upon.) Therefore, if your function needs to know the value of some variable, you **ought to pass that as a parameter**.

Here's an example of bad code. I have a account, which starts with principal P. Every now and then, I want to add some interest. I do this by supplying a value of r and and a value of t, using the formula

$$A = Pe^{rt}$$

-- or really,

$$\text{New value} = (\text{Old value})e^{rt}$$

I write a function to do this ... sort of.

In [ ]:

```
# EXAMPLE 1a: Compound interest
import math

val = float(input("Enter principal: "))

# This function is supposed to compute the accumulated value of an investment earni
ng compound interest.
# How can we change it to make it a work?
def accumulate_value():
    val = val*math.exp(r*t)


r = float(input("Enter annual interest rate as a decimal: "))
t = float(input("Enter time period in years: "))

accumulate_value() # Update P...??

r = float(input("Enter annual interest rate as a decimal: "))
t = float(input("Enter time period in years: "))

accumulate_value() # Update P...??

print(val)
```

To get this function to work, we need to have val, r and t be *inputs*, and the new accumulated value to be *returned* and then *assigned to val*.

In [ ]:

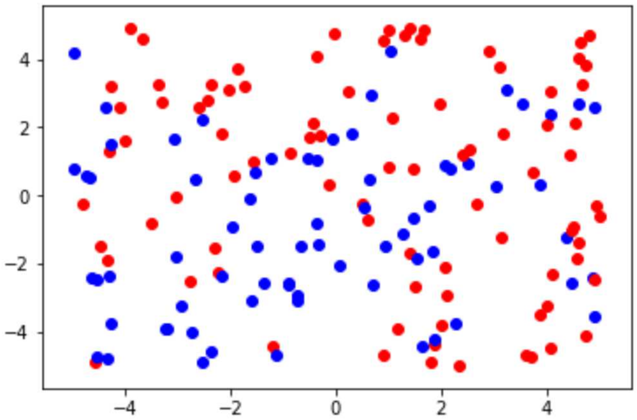
```
# EXAMPLE 1a': Compound interest
import math

val = float(input("Enter principal: "))

# CORRECTED: let's make P, r and t inputs.  Then, instead of updating P (which doesn't work anyway),
# return it.
def accumulate_value(r, t, val):
    return val*math.exp(r*t)


r = float(input("Enter annual interest rate as a decimal: "))
t = float(input("Enter time period in years: "))

val = accumulate_value(r, t, val) # Notice: not only are there inputs, but val gets updated!
print(val)
r = float(input("Enter annual interest rate as a decimal: "))
t = float(input("Enter time period in years: "))

val = accumulate_value(r, t, val)

print(val)
```

2. Making a Modular Program

In the town of Cartesian Plains, NY, every resident's address is given by an x and y coordinate. I have a data file called "surveydata.txt", which contains the location of many of the town's residents, as well as their preference for Coke or Pepsi.

I am scouting out locations to open a new deli, and I would like to know whether there are more Coke drinkers or Pepsi drinkers nearby, so that I can know what to carry. I would like a program where I can input an x-coordinate and a y-coordinate, and have as output the number of Coke drinkers and the number of Pepsi drinkers that are within a distance of 1 from the input point.

Just so that we can talk about it, here's a map I've made of the data: Coke is red, Pepsi is blue.



Python has tons of ready-made tools for the various aspects of problems like this, and I encourage you to learn about and use them! But remember, sometimes your particular problem won't match Python's built-in tool set, and you'll have to fashion your own tools. Right now, we're going to do this from scratch.

What subtasks will our code need to perform to solve this problem properly?

For instance:

- Read in the data, properly processing each row.
- Once all the data is processed, find the distance from each person to the input point.
- Identify which points are within 1 of the input point, and count the number of Coke drinkers and Pepsi drinkers among them.

The first two, at least, can definitely be written as functions.

- The first task could be accomplished by having a function that takes as **input** a file object, and **outputs** a list: each entry will be a list containing two `floats` and a `str`.
- The second task could be accomplished by having a function that takes as **input** four `floats`: two x coordinates and two y coordinates. The **output** should be the distance between them, as computed by the distance formula.

In [ ]:

```
# EXAMPLE 2a: Let's write the function that processes the file, and turns it into a
list.

def process_file(data_file_obj):
    """
    Take a file object as input.  Return a list containing its data.
    The list should have one entry per row, and each entry should contain 3 entries
:
    an x-coordinate, y-coordinate, and either 'Coke' or 'Pepsi.'
    """
    output_list = []
    for line in data_file_obj:
        values = line.split()
        # Remember to turn the coordinates to floats.
        values[0] = float(values[0])
        values[1] = float(values[1])
        output_list.append(values)
    return output_list

#
# Now, how could we test this?
#
sur_dat = open("surveydata.txt", "r")
data_list = process_file(sur_dat)
print(data_list)
sur_dat.close()
```

Note that it is a good idea to write the tests **before** writing the function. When you conceive of writing a function, you have in mind some task for it to accomplish. If it accomplishes something else instead, it's not meeting its responsibility.

Notice that this function should **return** the list -- it should **not print it**. This function takes a file and puts its data into a list. If you *print* the list, then it goes onto the screen, but that's it: *you cannot then do further computations with it.* If you *return* it instead, its output can be stored as a variable, and used later.

Let's write the distance function now.

In [ ]:

```
# EXAMPLE 2b: Distance function
# The parameters should be in the order
#        X1, Y1, X2, Y2 !!!

def distance(x1, y1, x2, y2):
    """Return the distance between (x1,y1) and (x2,y2) in the Cartesian plane"""
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

# TESTS:
print(distance(3,4, 0, 0), " (should be 5.0)")
print(distance(-1, 1, 2, 3), " (should be 3.605551)")
print(distance(4.2, 2.1, -1.3, 2.5), " (should be 5.514526)")
```

Here's a crazy idea: each row of the data is a list of the form `[x, y, "drink choice"]`, representing a resident. Let's make a function that takes the coordinates of the deli, and a row of data, and answers the question: is the resident within distance 1 of the deli?

In [ ]:

```
# EXAMPLE 2c: Is this resident close?

# Can you write the body in ONE line?
def is_close(deli_x, deli_y, resident):
    """
    Take the coordinates of the deli, and the data for a resident.  Return True if the distance
    between deli and the resident is less than 1.
    """
    res_x = resident[0]
    res_y = resident[1]
    if distance(deli_x, deli_y, res_x, res_y) < 1:
        return True
    else:
        return False

# Tests:
first_res = [1.1, 2.2, "Coke"]
second_res = [0.5, 3.3, "Pepsi"]
print(is_close(0.9, 2.5, first_res), " (should be True)")
print(is_close(0.9, 2.5, second_res), " (should be True)")
print(is_close(3, 3.5, first_res), " (should be False)")
```

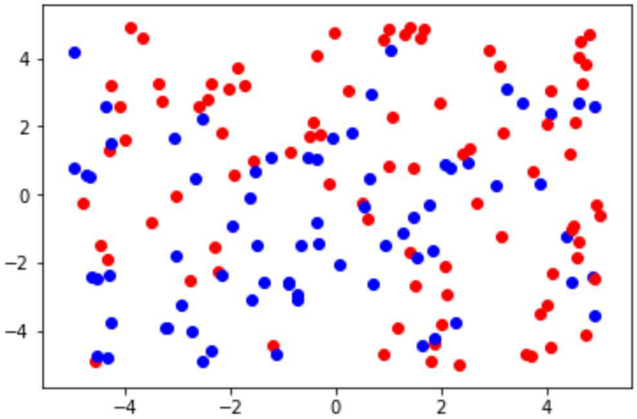
Then, we put the whole thing together. We define each function in the file before it is used.

Let's write our full program in a `main()` function. Among other things, this helps easily identify an "outline" of the entire program -- e.g. which line gets executed first, which line gets executed last, which other functions get called in which order.

In [ ]:

```
# EXAMPLE 2d: Now, let's put it all together.
# This program asks for a coordinate, opens the survey data, finds all resident within distance 1 from that point,
# and counts how many of those residents are Coke drinkers and Pepsi drinkers.

import math

def process_file(data_file_obj):
    """
    Take a file object as input.  Return a list containing its data.
    The list should have one entry per row, and each entry should contain 3 entries:
    an x-coordinate, y-coordinate, and either 'Coke' or 'Pepsi.'
    """
    output_list = []
    for line in data_file_obj:
        values = line.split()
        # Remember to turn the coordinates to floats.
        values[0] = float(values[0])
        values[1] = float(values[1])
        output_list.append(values)
    return output_list

def distance(x1, y1, x2, y2):
    """Return the distance between (x1,y1) and (x2,y2) in the Cartesian plane"""
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

def is_close(deli_x, deli_y, resident):
    """
    Take the coordinates of the deli, and the data for a resident.  Return True if the distance
    between deli and the resident is less than 1.
    """
    res_x = resident[0]
    res_y = resident[1]
    if distance(deli_x, deli_y, res_x, res_y) < 1:
        return True
    else:
        return False

######## MAIN FUNCTION ########
def main():
    # Open and process the survey data.
    survey_file = open("surveydata.txt", "r")
    data_list = process_file(survey_file)

    # Obtain the proposed location of the deli.
    deli_x = float(input("Enter x coordinate of deli: "))
    deli_y = float(input("Enter y coordinate of deli: "))

    # Perform the counts.
    coke_count = 0
    pepsi_count = 0
    for resident in data_list:
        # Recall: each resident is a list: [x-coord, y-coord, drink]
        # Only count the close residents.
        if is_close(deli_x, deli_y, resident):
            if resident[2] == "Coke":
                coke_count += 1
            elif resident[2] == "Pepsi":
                pepsi_count += 1

    # Final summary.
    print("Coke drinkers: {0}, Pepsi drinkers: {1}".format(coke_count, pepsi_count)
```

Here's that chart again, so we can test easily (red = Coke, blue = Pepsi):



And, as a bonus, here is the code that made the plot. Notice how we use the `process_file` function again. Also, remember matplotlib? We're using it again here.

In [ ]:

```
# EXAMPLE 2e: How I made the scatter plot

import matplotlib.pyplot as plt # Recall that matplotlib is a data visualization library.
                                # The " as plt " part is Python's way to help you shorten the name.

def process_file(data_file_obj):
    """
    Take a file object as input.  Return a list containing its data.
    The list should have one entry per row, and each entry should contain 3 entries:
    an x-coordinate, y-coordinate, and either 'Coke' or 'Pepsi.'
    """
    output_list = []
    for line in data_file_obj:
        values = line.split()
        # Remember to turn the coordinates to floats.
        values[0] = float(values[0])
        values[1] = float(values[1])
        output_list.append(values)
    return output_list

######## MAIN FUNCTION ########
def main():
    # Open file, process data.
    survey_file = open("surveydata.txt", "r")
    data_list = process_file(survey_file)

    # Plot each point in the scatter plot.
    for resident in data_list:
        if resident[2] == "Coke":
            # plt.scatter() is a function that adds a point to the plot.
            # The parameters are: x-coord, y-coord, shape of each dor, color of each dot.
            plt.scatter(resident[0], resident[1], marker = "o", color = "r")
        elif resident[2] == "Pepsi":
            plt.scatter(resident[0], resident[1], marker = "o", color = "b")

    # After all points are added, show the plot.
    plt.show()

###########################################
# This runs when we execute this program. #
main()
```

3. Lists (and other Mutables) as Arguments, and "Pass By Object Reference"

So far, we have passed numbers and strings to functions; they happen to be **immutable** data types (more precisely said, objects with these data types are immutable). However, if you pass a function an object whose data type is **mutable**, then you might be able to notice changes Python makes to the input.

In [ ]:

```python
# EXAMPLE 3a: A function that has SIDE EFFECTS.

def add_one(x, y):
    """
    The first parameter is a number; the second is a list.
    This function will 'change' both, but one of the changes you'll notice afterwards.
    """
    x = x + 1
    y[0] += 1

number = 5
num_list = [3,7,12]

add_one(number, num_list)
# The function has a SIDE EFFECT: it affects the value of the SECOND input,
# even though no further assignment has taken place outside of the function.
print(number, num_list)
```

A **side effect** of a function is a change to an actual parameter that occurs only due to assignments in the function. You won't ever notice them with immutable inputs, but they can occur when you *perform* ***modifications*** *to mutable inputs.*

The main mutable data types we've dealt with are lists and file objects. With these objects, you can perform modifications. For instance, suppose that `x = [3, 7, 12]`. If I were then to write

`x[0] = 1` or `x.append(5)`,

the object that x was associated with would change. On the other hand, if I were to write

`x = [1,2,3]`,

Python would create an entirely new list object, and assign that object to the variable `x`.

This matters because "modifications cause side effects, whereas assignments don't." I'll get to the whole truth in a moment, but let's see if we get this.

In [ ]:

```python
# EXAMPLE 3b: What side effects will take place from this function?

def fn(a, b):
    a[0] = "Hello"
    del a[1]
    a = ["Apple", "Banana", "Cantaloupe"]
    a[0] = "Goodbye"

    b = b + 1
    b = 5


first_in = ["Word", "Another", "Thirdword"]
second_in = 4

# Now apply the function.  What side effects occur?
fn(first_in, second_in)

# Print first_in and second_in when you have an idea.
```

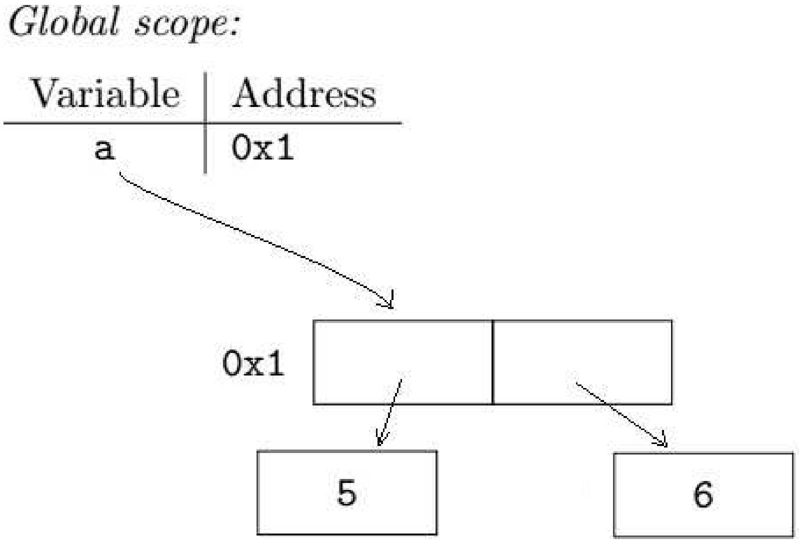
So, what is truly going on? Remember that variables are references to objects. When you call a function, the formal parameters become references to the same objects that are passed to them.

For example, consider the code

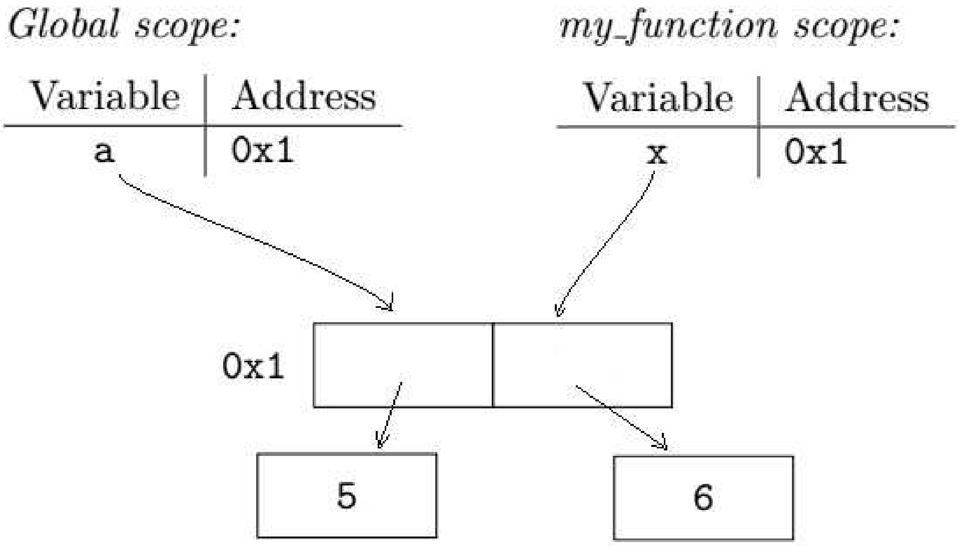
In [ ]:

```
# EXAMPLE 3c: A small example illustrating pass by object reference

def my_function(x):
    x[0] = 1
    x = [2, 3]
############################
a = [5, 6]
my_function(a)
print(a)
```

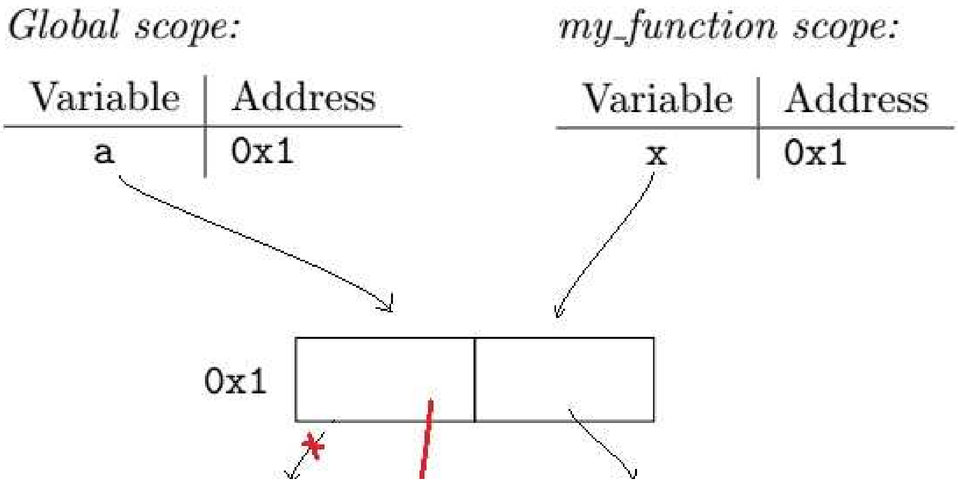
The line `a = [5, 6]` will create a `list` object with two entries, which `a` will point to.



The second line will call `my_function` with `a` as input; the local variable `x` will then be assigned to be a reference to the same `list` object.



The line `x[0] = 1`, as a modification line, will work directly with the `list` object.



In [ ]:

```
# EXAMPLE 3d: Insert in order

def insert_in_order(s_list, value):
    """
    Accept a sorted list (in increasing order), and a value to insert.  Insert the
value into the list,
    in the right position so that the list remains sorted.
    """

    for i in range(len(s_list)):
        # Insert the value at the FIRST position where
        # it is less than the value
        if value < s_list[i]:
            s_list.insert(i, value)
            break
        # If the value is not less than ANY of the elements
        # in the list: it should be placed at the end!
        if i == len(s_list) - 1:
            s_list.append(value)

################

x = [20, 40, 60, 80]
insert_in_order(x, 55)
print(x)
insert_in_order(x, 15)
print(x)
insert_in_order(x, 90)
print(x)
```