Lecture 6

If-Elif-Else; Nested If Statements; bool Variables; Print Debugging; Intro to Programmer-Defined Functions

1. If-Elif-Else Chains

If-else is great when you have two possibilites to choose from. What if you have 3 possibilities, or 5? In that case, you might want an ***if-elif-else chain***. ("Elif" is short for "else if".) Example:

In [ ]:
```
# EXAMPLE 1a: Pie

favorite = input("Please state your favorite type of pie: ")

if favorite == "Key Lime":
    print("Yeah, that's what's up!")
elif favorite == "Pumpkin" or favorite == "Apple":
    print("That's very American.  I can respect that.")
elif favorite == "Chocolate":
    print("You're not really a pie person, are you?")
else:
    print("Well, at least you didn't put \"Chocolate\".")
```

Here is the general syntax:

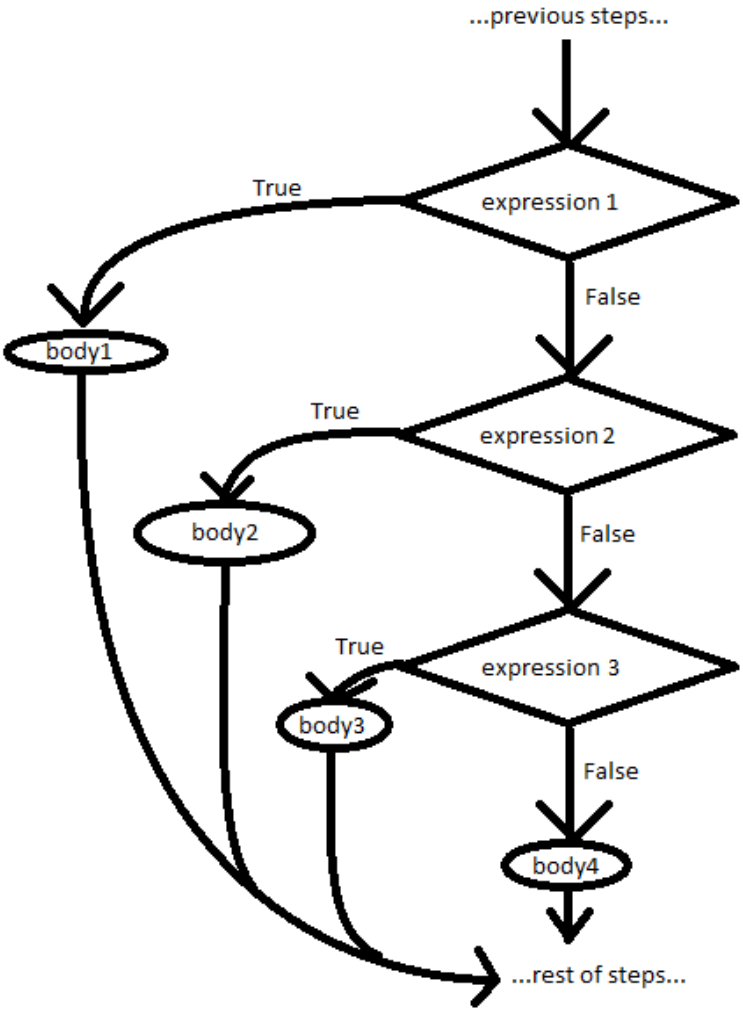
In [ ]:
```
IF-ELIF-ELSE CHAIN SYNTAX:

"...previous statements (unindented)..."

if <expr 1>:
    <body 1>
elif <expr 2>:
    <body 2>
elif <expr 3>:
    <body 3>
else:
    <body 4>

"...further statements (unindented)..."
```

Note that the `elif`'s go in the middle, and of course you can have more or less than 2. Also, the `else` at the end is optional.

The idea is that when this chain is encountered:

- The first logical expression, `<expr 1>`, is evaluated.
- If `True`, `<body 1>` is evaluated, and then execution skips past the rest of the chain.
- If `False`, then execution moves on to the first `elif`, and `<expr 2>` is evaluated. If this is `True`, then `<body 2>` is evaluated; if `False`, execution moves to the third `elif`.
- And so on. If there is an `else` at the end, that executes if none of the logical expressions are `True` (the "default" scenario); if there is no `else` at the end, then it is possible for none of the bodies to execute.

Here's a flowchart, for an if-elif-else chain with an else at the end:



Again: notice that **exactly one** of the bodies is executed. (And if there were no `else`, then **at most one** body would be executed.)

Write a program which asks the user to put in a lowercase letter. Then have it print out `Always` if it is a,e,i,o,u; `Sometimes` if it is y; and `Never` otherwise. (To test this properly, you need a bare minimum of three test runs.)

In [ ]:
```
# EXAMPLE 1b: Vowels
# Ask user to input a lowercase letter, then print out whether or not it is
# a vowel always, sometimes, or never

letter = input("Enter a lowercase letter: ")

if letter == 'a' or letter == 'e' or letter == 'i' or letter == 'o' or letter == 'u
':
    print("Always")
elif letter == 'y':
    print("Sometimes")
else:
    print("Never")
```

I have here three versions of a program. The program is a (slightly lazy) grade converter. It asks for a score, and then prints out A for 90-100, B for in the 80's, a `C or lower` for less than 80. All versions have flaws. What are the flaws, exactly?

In [ ]:
```
# EXAMPLE 1c: Flawed Grades, Version 1

score = input("Enter score: ")
score = float(score)

if score >= 90:
    print("A")
else score >= 80:
    print("B")
else:
    print("C or lower")
```

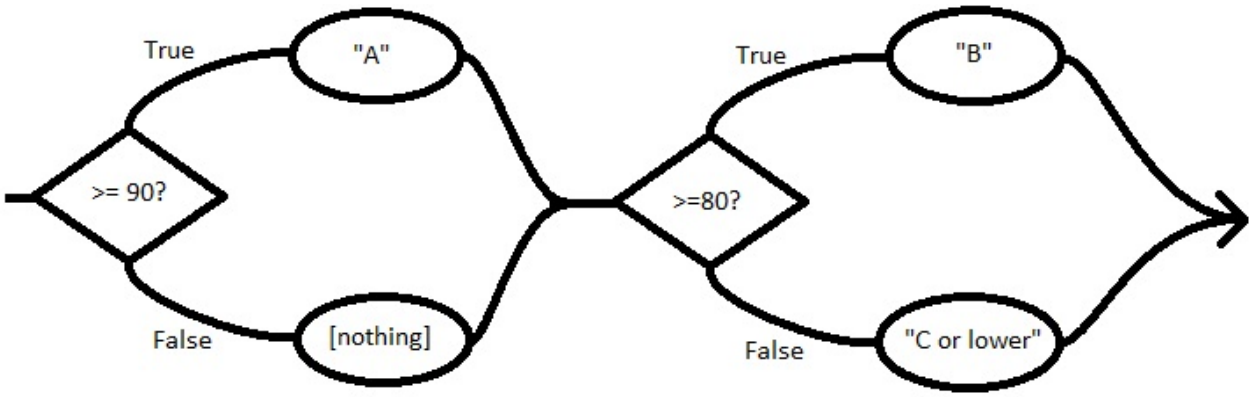
In [ ]:
```
# EXAMPLE 1d: Flawed Grades, Version 2

score = input("Enter score: ")
score = float(score)


if score < 80:
    print("C or lower")
elif score >= 80:
    print("B")
elif score >= 90:
    print("A")
```

In [ ]:
```
# EXAMPLE 1e: Flawed Grades, Version 3

score = input("Enter score: ")
score = float(score)

if score >= 90:
    print("A")
if score >= 80:
    print("B")
else:
    print("C or lower")
```

Version 1: This is a straight syntax error: there is no such thing as `else {logical expression}`. `else` is never followed by anything but a `:`. Obviously, `elif` was intended.

Version 2: The backwards order isn't inherently a problem, nor is the lack of an `else`. What IS a problem, however, is that a `score` that is `>= 90` will also be `>= 80`, so `B` will print out, and you will never get to the second `elif`.

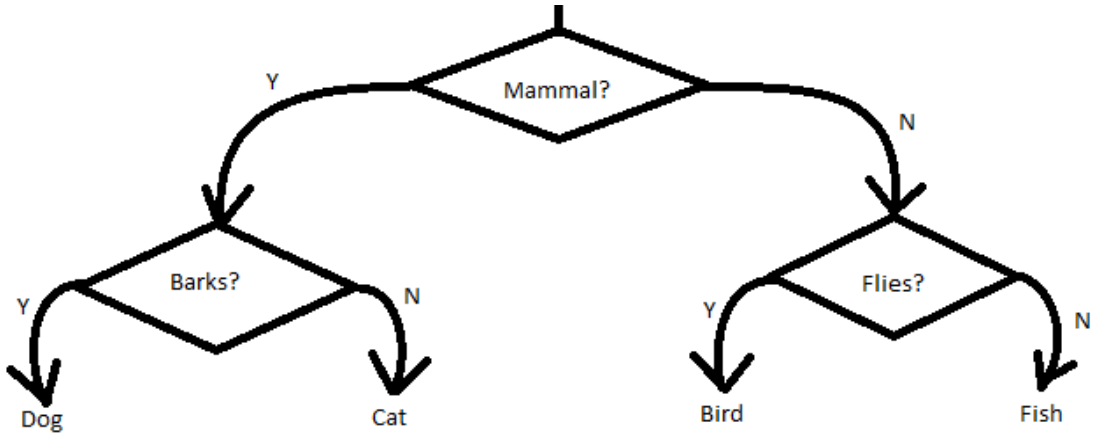
Version 3: Notice that the middle statement is `if`, not `elif` -- so the first two lines are severed from the last four! So if `score >= 90`, then `A` prints out; then, you execute the next if-else statement: for which `B` will also print out. Here's a flow chart illustrating:



2. Nested If Statements and Pseudo-code

There's no reason you can't put if (or if-else) statements inside one another. They frequently come up when, in an if statement, the `True` branch is significantly different than the `False` branch.

Example: 2 questions! The program we're about to write will allow the user to think of an animal; the program then asks questions, which the user answers, and based on that it tries to read your mind. (It doesn't do a very good job, unless you happen to be thinking of a dog, cat, bird or fish -- in which case it works!)



Translate from the top down.

Specifically, start by looking at the top question. **Everything** coming out of the top "yes" arrow should be inside the `if` block; and **everything** coming out of the top "no" arrow should be inside the `else` block. So, we're going to have an `if` statement inside a block! To make this work, we'll have to indent twice. Let's look at the code.

In [ ]:
```
# EXAMPLE 2a: 20 questions

print("Think of an animal, then press Enter.")

# Here is a trick to make your program have a dramatic pause.
input("")

print("I'm going to try and guess it now.")

input("")

first_ans = input("Is it a mammal? ")


if first_ans == "y":
    second_ans = input("Does it bark? ")   #
    if second_ans == "y":                  #  These 5 lines are
        print("It's a dog!")               #  the "if" block for the
    else:                                  #  first "if-else" statement.
        print("It's a cat!")               #
else:
    second_ans = input("Does it fly? ")    #
    if second_ans == "y":                  #  These 5 lines are
        print("It's a bird!")              #  the "else" block for the
    else:                                  #  first "if-else" statement.
        print("It's a fish!")              #
```

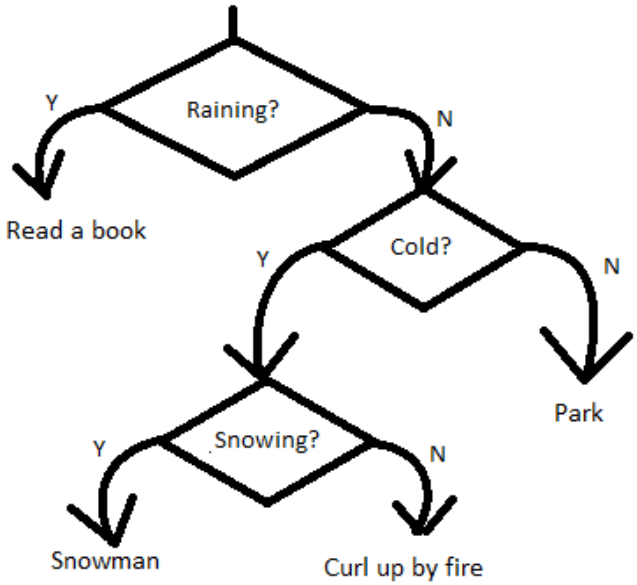
When designing a complex program, it's helpful to write down ***pseudo-code*** before sitting down to your computer. Pseudo-code means the steps of the program written down in English, arranged to resemble source code, but without worrying about the details inherent in programming correct steps -- like proper variable names, data types, etc.

For this flowchart:

In [ ]:
```
Ask if it is a mammal
If it is a mammal:
    Ask if it barks
    If it barks:
        It is a dog
    Otherwise:
        It is a cat
Otherwise (if it is not a mammal):
    Ask if it flies
    If it flies:
        It is a bird
    Otherwise:
        It is a fish
```

In both real code and pseudo-code, **indenting** properly is crucial. Real Python code won't work right if you don't indent properly; with pseudo-code, indenting shows which lines are grouped together, and which alternatives are connected to which.

Now, code this flow chart!



In [ ]:
```
# EXAMPLE 2b: Weather

first_ans = input("Raining? ")
if first_ans == "y":
    print("Read a book")
else:
    second_ans = input("Cold? ")
    if second_ans == "y":
        third_ans = input("Snowing? ")
        if third_ans == "y":
            print("Snowman")
        else:
            print("Curl up by fire")
    else:
        print("Park")
```

3. Using `bool` Variables

Suppose that you need to keep track of a LOT of decisions.

For example: suppose you want to check that three sides of a triangle, a, b and c, are entered in descending order, are all greater than 0, and satisfy the triangle inequality (that the sum of any two sides is always greater than the third)?

Here is one way to code this:

In [ ]:
```
if (a >= b) and (b >= c) and (a > 0) and (b > 0) and (c > 0) and (a + b > c) and (a
+ c > b) and (b + c > a):
    # Solve triangle
else:
    # Print error message
# (Let's ignore the fact that for mathematical reasons, there's some redundancy in
the above.)
```

But that's a lot for a single line (and I'm sure you could imagine that it could get way worse). Fortunately, you can use `bool` variables (also known as ***flags***) to help you deconstruct such a statement.

Here's an alternative, more digestible way to deal with this statement, that breaks the decision process down. Notice also how the `if` statements almost read like honest English sentences -- don't underestimate how useful this can be when you are deep in the weeds trying to read complex code.

In [ ]:
```
in_order = (a >= b) and (b >= c)
all_positive = (a > 0) and (b > 0) and (c > 0)
satisfies_triangle_ineq = (a + b > c) and (a + c > b) and (b + c > a)

if in_order and all_positive and satisfies_triangle_ineq:
    # Solve triangle
else:
    if not in_order:
        print("Out of order")
    if not all_positive:
        print("Non-positive side entered")
    if not satisfies_triangle_ineq:
        print("Your sides don't satisfy the triangle inequality")
```

Some people don't like seeing `if` statements without =='s or >'s or the like, and freak out. *Don't freak out -- evaluate*! Evaluate the whole logical expression, like before. If the expression evaluates to `True`, then the statement prints; if it evaluates to `False`, then the statement doesn't print.

The next program is meant to allow the user to enter 4 names, and then at the very end print out whether or not, at some point, the same name was entered twice in a row (at least once -- if a name was entered three times in a row, or two different names were each entered twice in a row, we'll include that).

Since we'll need the answer to a yes or no question at the end, we'll use a boolean variable to store the answer.

In [ ]:
```
# EXAMPLE 3a: Same Name Twice In A Row
# We may need to fix a couple of things.

# repeat_yet will contain the answer to the question: "have two identical names bee
n entered consecutively yet?"
# What should repeat_yet be initialized to?
repeat_yet =


name1 = input("First name: ")
name2 = input("Second name: ")

if name1 == name2:
    repeat_yet = True
else:
    repeat_yet = False

name3 = input("Third name: ")

if name2 == name3:
    repeat_yet = True
else:
    repeat_yet = False

name4 = input("Fourth name: ")

if name3 == name4:
    repeat_yet = True
else:
    repeat_yet = False


if repeat_yet:
    print("There were two consecutive entries that were the same.")
else:
    print("No consecutive repeats.")
```

First, let's talk about initializing `repeat_yet`. In the beginning (that is, at time of initialization), what is the status of `repeat_yet`? Have we made encountered a repeat yet? No! So I think this should be set to `False` in the beginning. (You could write a program like this with a boolean variable that is initialized to `True`, but I think that would be trickier.)

Now, what happens if you enter `Alice`, `Jo`, `Jo`, `Bob`? Whoops, the program says that there are no consective repeats. Why? Let's look at the last `if` statement. Before going in, `repeat_yet` was `True`. But then, since `name3` does not equal `name4`, `repeat_yet` gets set back to `False`! That shouldn't happen -- we **have** seen a repeat, it's just that it occurred *earlier*.

The key is that `repeat_yet` should start out as `False`, but it should under no circumstances be set back to `False` once it becomes `True` -- once there's been a repeat, this variable should remain `True` for the rest of the program. The code as written above frequently assigns `repeat_yet` to be `False` -- improperly.

Fortunately, there's a simple fix here: chop off the first three `else`s (the one at the very end should remain, of course). Not every `if` needs an `else` -- and here, the `else`s were actively harmful!

In [ ]:
```
# EXAMPLE 3a': Same Name Twice In A Row
# We may need to fix a couple of things.

# repeat_yet will contain the answer to the question: "have two identical names bee
n entered consecutively yet?"
# What should repeat_yet be initialized to?
repeat_yet = False


name1 = input("First name: ")
name2 = input("Second name: ")

if name1 == name2:
    repeat_yet = True

name3 = input("Third name: ")

if name2 == name3:
    repeat_yet = True

name4 = input("Fourth name: ")

if name3 == name4:
    repeat_yet = True

######

if repeat_yet:
    print("There were two consecutive entries that were the same.")
else:
    print("No consecutive repeats.")
```

Challenge: can you rewrite the code above the ###### without any `if` statements, so that the program works properly, without changing the code below ######?

4. Print Debugging

As you can probably tell, our programs are getting more complicated. At this point, we have to get better at debugging. The simplest, most direct way is by using **walkthroughs** and **print statements**.

When you write a program, (hopefully) you have an idea of what you think each line is doing to the values of your variables. A walkthrough can give you confirmation or denial of your suspicions.

It helps to include print statements throughout your program to print out the values of certain variables, to confirm whether your suspicions are correct at each line. If one is not: that's going to give you a big hint about where your error is.

Let's see this in action. The code below is **meant** to do the following. It asks for two numbers from each of two players. It then adds up each player's numbers, and outputs which player is the winner, using the following rules:

Any player with a sum > 21 is disqualified. The winner is the player with the highest sum who is not disqualified. If both players are disqualified, then no one wins. If both players have the same sum which is <= 21, then the players both win.

(So, this is a lot like the game of Blackjack, if you know what that is.)

Note that the following isn't really good code, but I promise you that the general idea is fine -- there's just an error in implementation. Can we find it?

In [ ]:
```
# Example 4a: "Blackjack" (kind of)

p1_num1 = int(input("Enter Player 1's first number: "))
p1_num2 = int(input("Enter Player 1's second number: "))
p2_num1 = int(input("Enter Player 2's first number: "))
p2_num2 = int(input("Enter Player 2's second number: "))

# First, determine who has the higher sum
if p1_num1 + p1_num2 > p2_num1 + p2_num2:
    # Player 1 has the higher sum.
    # If that sum is 21 or under, player 1 wins.
    # If not, player 2 wins, unless player 2's sum is also over 21,
    # in which case no one wins.
    if p1_num1 + p1_num2 <= 21:
        print("Player 1 wins.")
    elif p2_num1 + p2_num2 <= 21:
        print("Player 2 wins.")
    else:
        print("Both players disqualified.")

elif p2_num1 + p2_num2 > p1_num1 + p1_num2:
    # Player 2 has the higher sum.
    # If that sum is 21 or under, player 2 wins.
    # If not, player 1 wins, unless player 1's sum is also over 21,
    # in which case no one wins.
    if p2_num1 + p2_num2 <= 21:
        print("Player 2 wins.")
    elif p1_num1 + p2_num1 <= 21:
        print("Player 1 wins.")
    else:
        print("Both players disqualified.")

else:
    # Both players have the same sum.  Both win if that sum is 21 or under
    # and both are disqualified otherwise.
    if p1_num1 + p1_num2 <= 21:
        print("Both players win.")
    else:
        print("Both players disqualified.")
```

Try putting in 10 and 2 for the first player, and 20 and 4 for the second player. Player 1 should win! That's not what happens, though: why?

See the typo in line 32? It should be p1_num1 + p1_num2.

5. A Quick Introduction to Programmer-Defined Functions

We have used a lot of functions in Python: `len()`, `print()`, `math.exp()`, `random.randrange()`, etc. As a programmer, you can write your own functions, too!

Why would you want to? Because **functions can help you break down your problems into smaller, more manageable parts** and **minimize repetition of code**. The act of writing your programs in pieces is sometimes called *modular programming*.

Before justifying that this really is useful, let's look at an example, which illustrates the mechanics (if not the purpose) of programmer-defined functions.

In [ ]:
```
# EXAMPLE 5a: A programmer-defined function
# This function computes sums of squares of numbers.
# Presumably, this is something that we may find ourselves doing a lot.

def sum_of_squares(x, y, z):
    sos = x**2 + y**2 + z**2
    return sos
```

Run that code. It doesn't appear to really do much! But it defines a new function. The keyword `def` denotes that we're defining a function; `sum_of_squares` is the *name* of the function; `x`, `y` and `z` represent the *inputs* to the function; and the word `return` denotes that the *output* of the function is the value called `sos`.

Now, let's **use** the function.

In [ ]:
```
# EXAMPLE 5b: Using that function

def sum_of_squares(x, y, z):
    sos = x**2 + y**2 + z**2
    return sos

###################################
# Here's where the action starts.

# On this line, I am CALLING (using) the function.
# The variables x, y and z temporarily become the inputs 1, 3, 5, respectively.
# When you USE the function, you don't write "def"
# (the same way that you haven't written def when you use len() or math.sqrt())
a = sum_of_squares(1,3,5)
print(a)

# A function is meant to be reused!  I don't rewrite the definition of the function
,
# but I use it multiple times.
b = sum_of_squares(1,4,0)
print(b)

# Of course, these three lines happen to do nothing, for exactly the same reason...
3 + 5                  #
len("Hello")           # They all compute values, which then go unused.
sum_of_squares(1,1,1)  #


# You can use variables and expressions as inputs, as well.
p = 4
q = 1
print(sum_of_squares(p, 2*q, p + q))
```

Let's go through this. There are two things to understand -- how the function is *defined*, and how it is used (or *called*). Let's start with the definition.

In [ ]:
```
BASIC FUNCTION DEFINITION SYNTAX:

def <function name>(<parameter list>):
    <body>
    return <output value>
```

The word `parameters` refers to names used for the inputs when you define the function. So in the program above, the parameters were `x`, `y` and `z`, and the output value was called `sos`.

When your program encounters a function definition, it does .... well, not much, at first. This is just a *definition* of a function -- what you see above the ################## in this example.

But then, later, you may *call* this function, which would look something like

`{function name}(<actual parameters>)`

Calls to a function must occur lower in the code than where that function is defined. In our program, the calls looked like `sum_of_squares(var1, var2, var3)`. When Python encounters `sum_of_squares(var1, var2, var3)`, program control is passed to the function -- that is, the function will stop its normal execution, and instead executing the function's code. More specifically, here is what happens:

- The parameters of the function will be matched with the actual parameters (or *arguments*). This means that x will be assigned the value of `var1`, y will be assigned the value of `var2`, and z will be assigned the value of `var3`. (First gets matched with first, second gets matched with second -- order matters here.)
- The body of the function will execute, using these values.
- The line `return sos` ends the function's execution, and the value of `sum_of_squares(var1, var2, var3)` will be whatever `sos` is.
- That value then gets stored to a variable, or printed, or ignored, depending on what you code does with `sum_of_squares(var1, var2, var3)`, and the program continues.

Let's write one together. (This would be a very useful function to write, were it not actually already part of the Python language.)

In [ ]:
```
# EXAMPLE 5c: Max function
# Let's write a function called my_max.
# This function should receive two arguments, are output whichever is greater.

def my_max(first, sec):
    if first > sec:
        output = first
    else:
        output = sec
    return output

a = 3
b = 4
c = 8

print("my_max(a,b) =", my_max(a, b))
print("my_max(c+1,b) =", my_max(c + 1, b))
print("my_max(5,c) =", my_max(5, c))
```

Try writing a function called `my_abs`, which should receive one numerical argument, and return the absolute value of that argument.

In [ ]:
```
# EXAMPLE 5d: my_abs

def my_abs(x):      # ALTERNATIVE BODY:
    if x > 0:       # if x > 0:
        ab = x      #     return x
    else:           # else:
        ab = -x     #     return -x
    return ab

print("If this isn't 2, your function has a problem :", my_abs(2))
print("If this isn't 4, your function has a problem :", my_abs(-4))
print("If this isn't 0, your function has a problem :", my_abs(0))
```