Assignment 2

Release date: Jan 13, 2023
Submission deadline: Jan 17, 2023

Source: Cornell (Intro to Computing With Python)

This is a simple written assignment where you are going to diagram a few call frames. The assignment is **to be attempted in groups of three**. You are allowed to discuss within your own group, but with nobody outside of the group. Each group member is supposed to contribute to the solution. There may be a following viva.

“Diagramming” is what we do in the lecture notes where we draw boxes to represent function calls and folders to represent the objects. These are a way of giving us a visual representation of what is going on in the computer when it runs Python. The purpose of this assignment is to solidify your understanding of how function calls work.

Learning Objectives

This assignment is design to help you understand the following concepts.

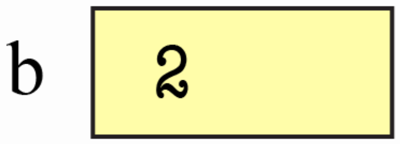
- The use call frames to model function execution.
- The difference between a call frame and global space.
- The use of folders to represent Python’s handling of objects.

Diagramming Conventions

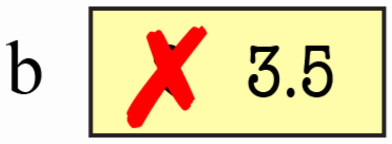
Most of our diagramming conventions follow the lecture notes. However, just to be safe, we will make our conventions explicit here.

Diagramming Variables

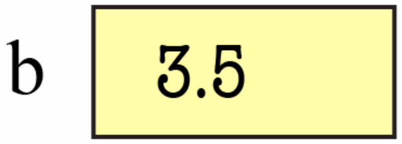
A diagram for a variable should include the variable name and a box for the value. Here is a diagram for the variable `b`, currently holding the integer 2:



When we reassign a variable, we cross out the old value and replace it with the new value. For example, if we reassign `b` to 3.5, we get



You only need to do this at the step it changes. In future pictures, we can drop the 2 and just use 3.5 as follows:



When writing the value, the only rule is to make sure it is unambiguous what the value is, and what type it is. Write floats with a decimal point or in scientific notation, write strings with quotes around them, and so on.

Diagramming Objects

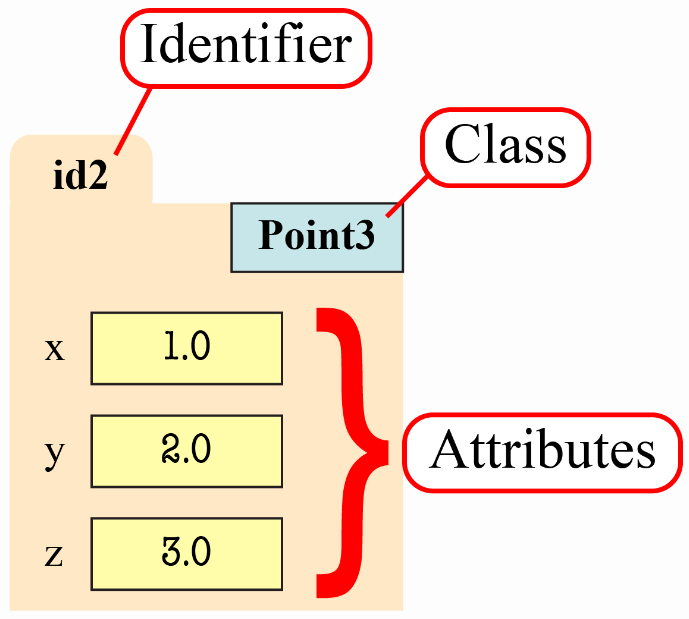
All values in Python are objects, but we only bother to draw an object separately from variables referring to it if the object is *mutable* (e.g. the contents of the folder can be changed). All of the basic types – `int`, `float`, `bool`, and `str` – are immutable. Therefore we draw all of these values inside the variable referring to them.

So far the only mutable objects we have seen are those with named attributes, like those of type `Point3`.

You will not be responsible for figuring out how to diagram an object of an arbitrary class. For any mutable object (which requires a folder), we will show you how to diagram all objects of its class. When you diagram an object you should obey the following conventions:

- The folders identifier (ID, or true name) should be unique and go on the folder tab.
- The class (type) of the object should go at the top right of the folder.
- The attributes should be listed inside the folder and diagrammed just like variables.

For example, a folder for an object of the type `Point3` demonstrated in Module 11 might look like this:



Diagramming Function Calls

Call frames are the way we diagram the temporary workspace for variables used during a function call. Diagrams of frames should obey the following conventions:

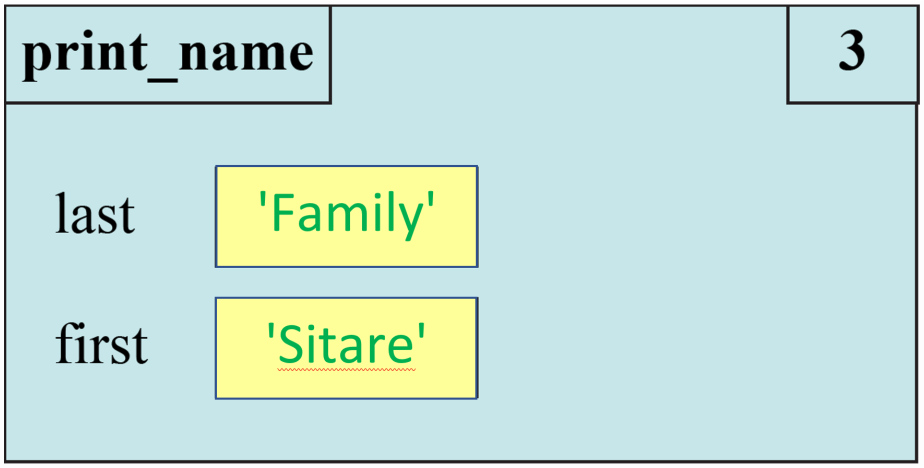
- The top left corner is a box with the function name.
- The top right is the program counter, the **next line of code** to execute.
- The instruction counter should start at the first line of *executable* code in the body.
- Local variables and parameters are both drawn in the frame body.
- Local variables are not added until their assignment statement **finishes** executing.
- Otherwise there is no distinction made between local variables and parameters.

See the lecture notes for more details on how to create a call frame at the start of a call. To help you with the instruction counter, we will always number the lines of code for you. For example, suppose we have the procedure

```
1   def print_name(last, first):
2       """Prints a greeting to 'first last' """
3       greet = 'Hello '+first+' '+last+'!'
4       print(greet)
```

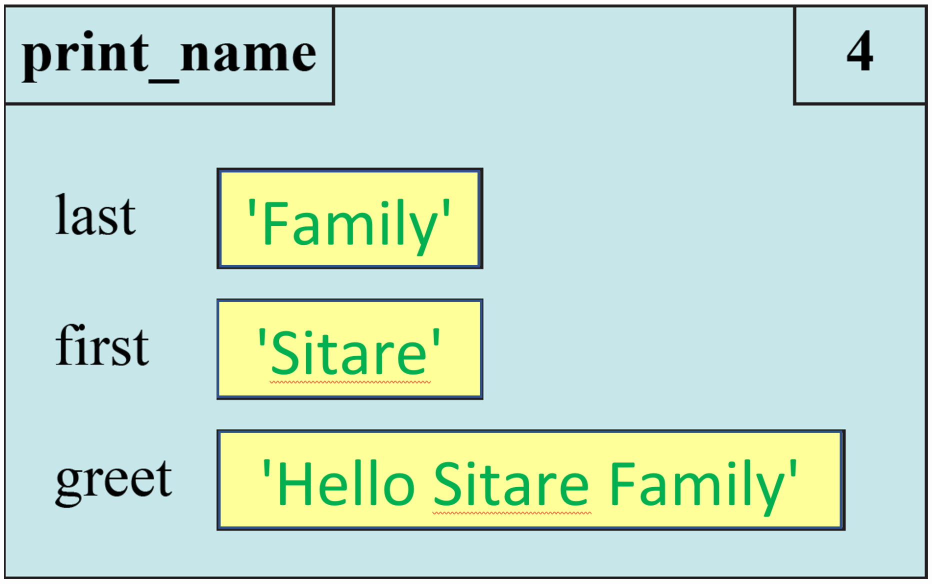
If we execute the call `print_name("Sitare","Family")`, then the call frame starts out as follows:

Notice the initial line number is 3. Line 2 is the docstring and hence is not an executable line of code.



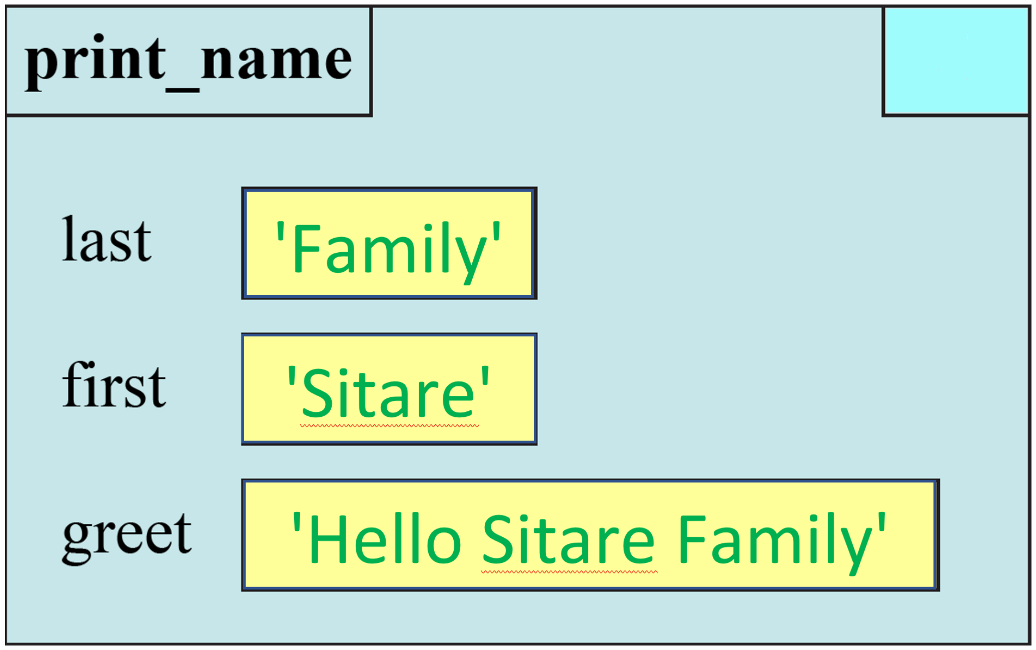
The Evolution of a Call Frame

A call frame is not static. As Python executes the body of the function, it alters the contents of the call frame. For example, after we have executed line 3, the call frame has changed to look like

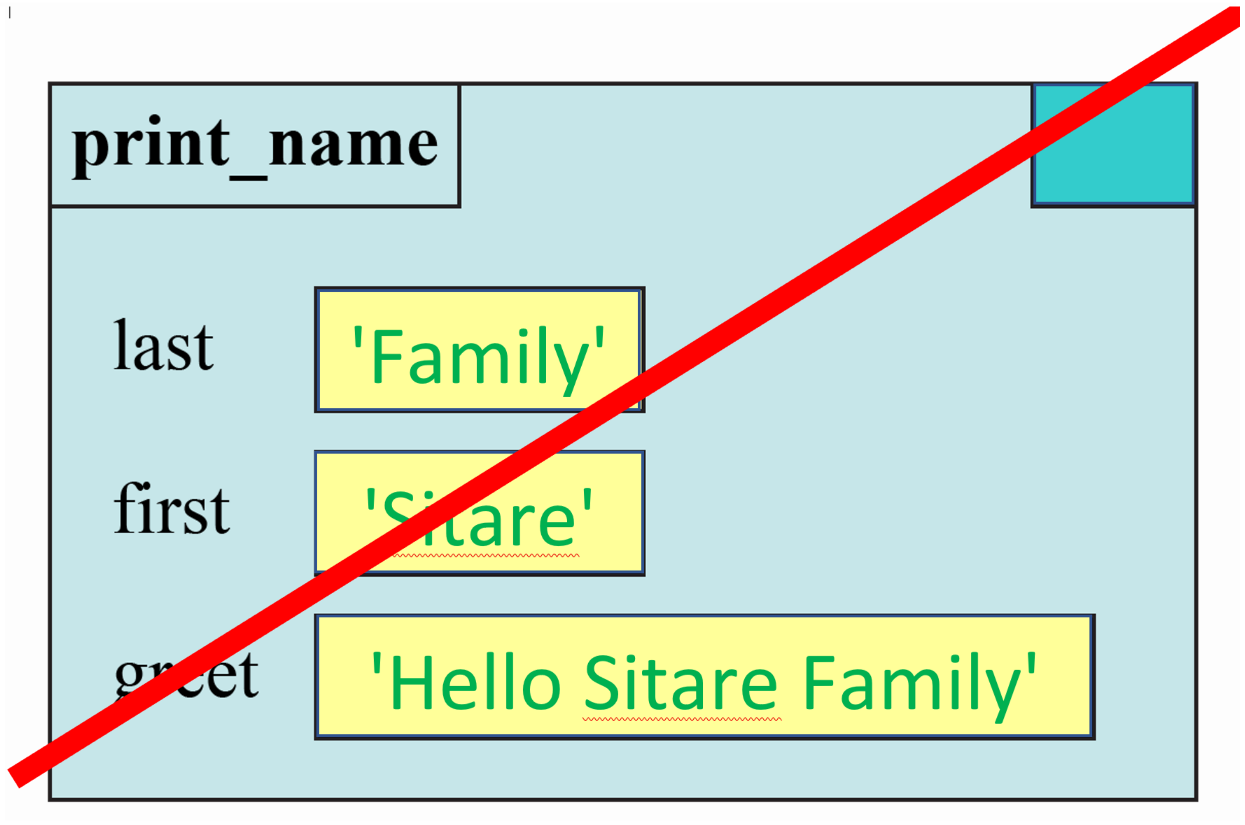


This is not a new call frame. This is the same call frame shown at a different point in time. The call frame does not go away until the function call is completed. When we diagram a call frame, we are actually diagramming a *sequence* of boxes to show how it changes over time. It is the same idea as creating an animation as a sequence of still images.

As a general rule, we should draw a separate frame diagram for each value of the instruction counter, corresponding to a single line of Python code. The diagram above refers to the call frame after we finish line 3. Once we finish line 4, the diagram for the call frame is as shown below.



Note the instruction counter is blank. That is because there are no more instructions to execute. However, the call frame is not yet deleted. To represent the deletion of the call frame, we draw one more diagram, crossing out the frame.



Frames and Return Values

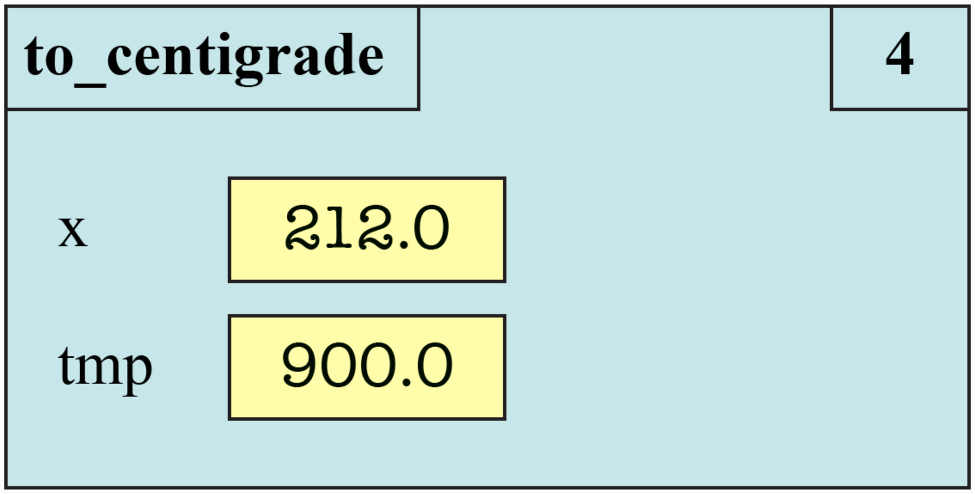
Remember from class that *fruitful functions* have a return value. For example, consider the following variation on a function from the lesson videos.

```
def to_centigrade(x):
    """Return the value of x in centigrade"""
    tmp = 5*(x-32)
    return tmp/9.0
```

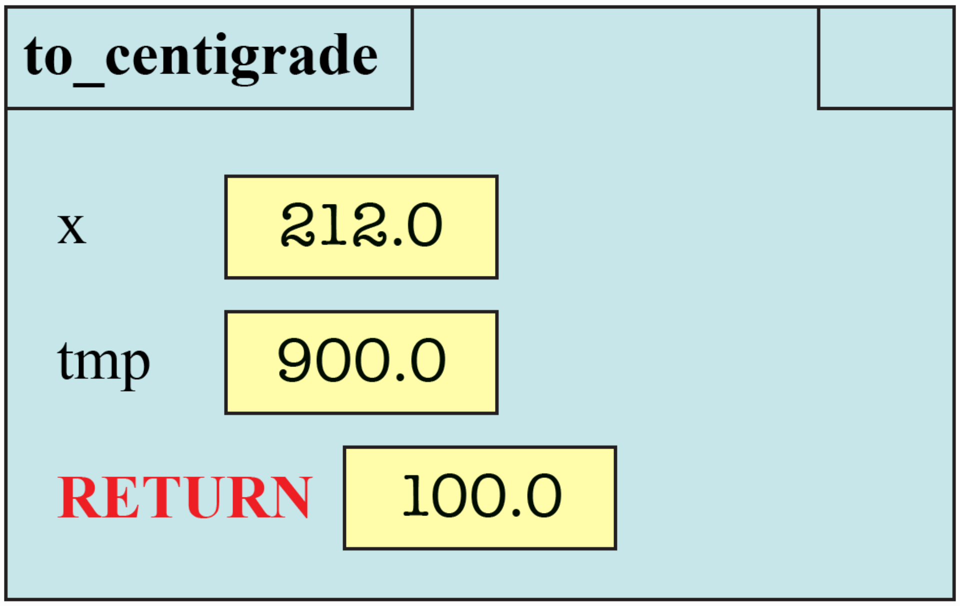
When you execute a return statement, it makes a special variable called `RETURN` which stores the result of the expression. For example, suppose that we have the function call

```
to_centigrade(212.0)
```

After executing line 3 of the code above, the call frame is as follows:



When you execute the next line, the instruction counter becomes blank (because there is nowhere else to go), but you add the `RETURN` variable.



Remember that **you should only add a `RETURN` variable for fruitful functions, and not for procedures**.

Frames and Global Space

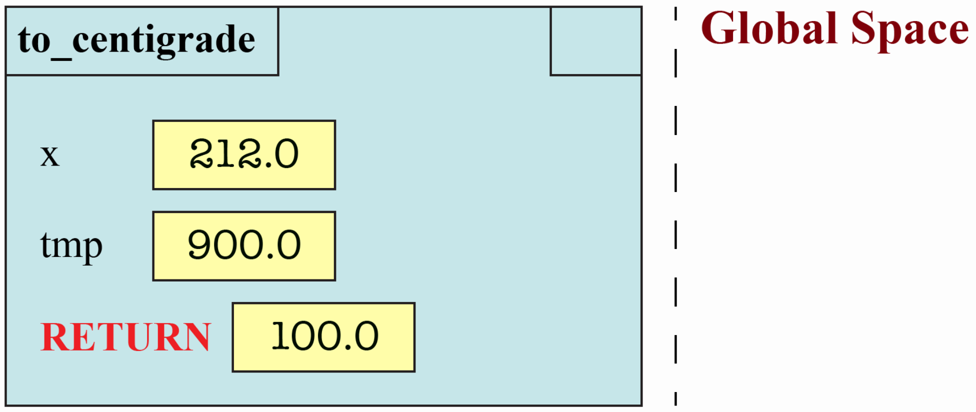
Remember that you can use a function call as an expression. The call evaluates to the result of the `return` statement. If this call is used in an assignment statement, this will copy from the `RETURN` variable to another place in memory (such as global space). For example, the call

```
    y = to_centigrade(212.0)
```

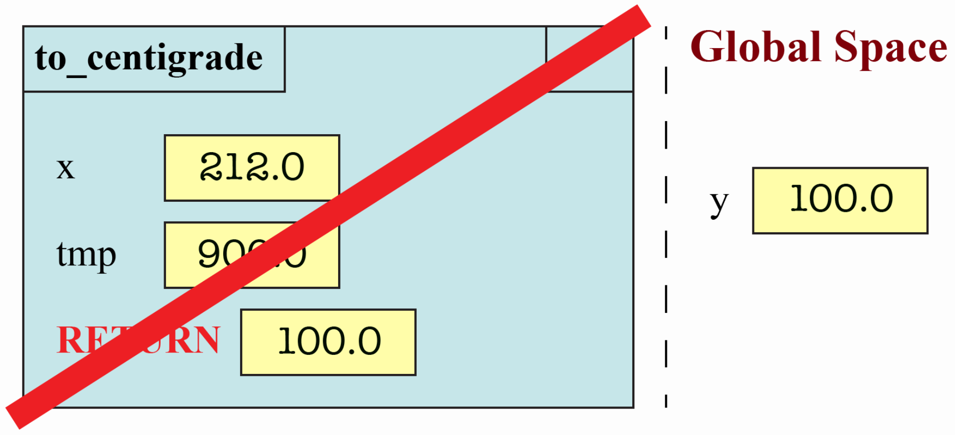
will copy the final result (`100.0`) into the global variable `y`.

If function call is used in an assignment statement, then the value in the box is not changed until the step where you erase frame. In fact, if this is the first time that you are assigning the variable, you do not even create the box for the variable until the frame is erased.

For now, you should denote changes to global space right next to the diagram for the frame. In our example above, that your diagram before erasing the frame is as follows:



When you do erase the frame, the global space has changed to



Modules and Functions in Global Space

As we saw in Module 14, global space also has variables for the function and module names. For example, when you define a function, it creates a folder for the function body and stores the name of that folder in a variable with the same name of the function.

We do not want you to draw these variables in global space. You can ignore what happens to function definitions (and module importing) for now.

Assignment Instructions

Remember that there are three memory locations that we talked about in class: global space, call frames, and the heap (where the folders live). Every time that you execute a line of code, these can all change. We want you to draw pictures to show how they change over time.

For each problem, you will be given a function call. We want you to diagram the execution of four different function calls (one each in Part **A**, **B**, **C** and **D**). You will draw a separate diagram **for each line of Python** that is executed (and only those lines that are executed). We also want you to draw one last diagram crossing the frame off at the end.

Some of the problems below will have associated values in both global space and the heap. For those problems, you should also diagram the evolution of both global space and the heap. For each line of code, next to the diagram for the call frame, you should redraw both global space and the heap. If there are no changes to these areas of memory, you are allowed to write UNCHANGED.

Important: You must pay close attention to types in this assignment. The value `3` is an integer while the value `3.0` is a float. Hence they are different values. If you answer with an integer like `3` when the correct answer is a float like `3.0`, we will deduct points.

Function `max`

The function `max` is built into Python. However, if we were to design it ourselves, it might look like this:

```python
def max(a,b,c):
    """Returns: maximum of a, b, and c

    Precondition: a, b, c are numbers"""
    m = a
    if (c > b and c > a):
        m = c
    elif b > a:
        m = b
    return m
```

Part A: Diagram the Execution of `d = max(1,2,3)`

Diagram the execution of the assignment statement

```python
d = max(1,2,3)
```

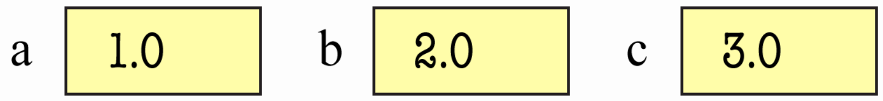
You will need to draw the evolution of the call frame over time. This will require several diagrams. You will need a diagram when the frame is created, and another when it is erased. In between, you will need a diagram for each line of code that is executed (and *only* the lines which are executed). Remember, this is one call frame, but several diagrams showing how the call frame changes over time.

When an assignment changes the value of an existing variable in the frame, we would like you write the old value in the box, cross it out, and write the new value in the box. However, **you are allowed to remove the crossed out value in later drawings of the frame.**

In addition to the diagrams for the (evolution of the) call frame, you also need to diagram the evolution of the global variable `d`. In particular, global space starts out empty (we are ignoring the variable for the function `max`). We want to know when the variable `d` appears in global space. You **do not** need to draw the evolution of the heap, as there is nothing interesting in the heap for this problem.

Part B: Diagram the Execution of `e = max(c,b,a)`

For this next part, suppose that you have the following (global) variables:



As we said in class, global variables and are *not* the same as the parameters of `max`, which are local to that function.

Repeat the previous exercise for the assignment statement

```
   e = max(c,b,a)
```

Once again, you need to draw the evolution of global space right next to each individual frame diagram. The difference is that this time global space does not start out empty. In addition, the new global variable is now named `e`.

Functions on `Vector2` Objects

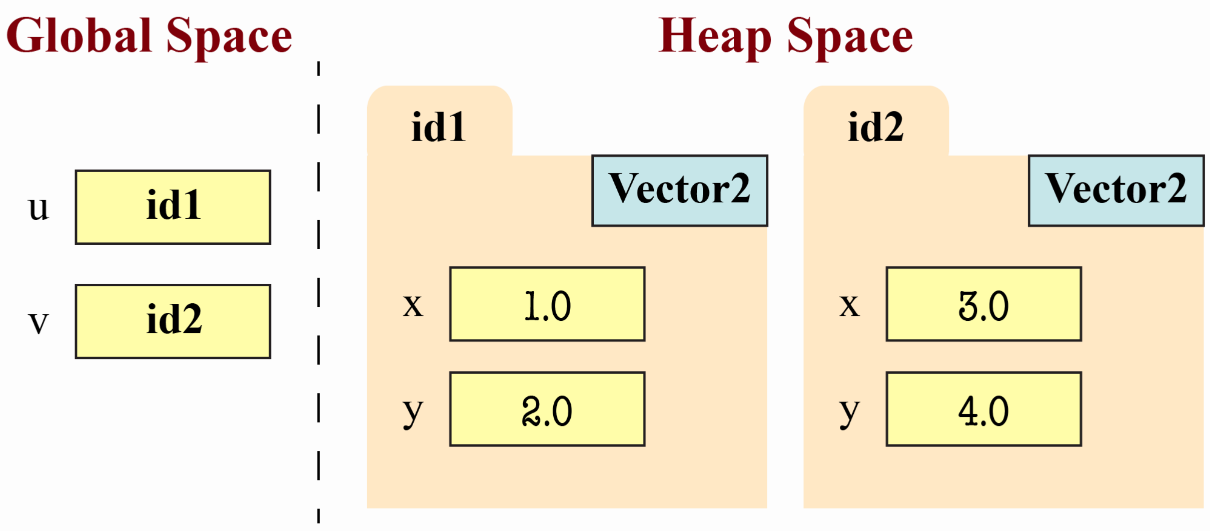
The next two problems have to do with the class `Vector2` class. This class is very similar to the `Point3` class introduced in lecture except that `Vector2` objects have only two attributes: `x` and `y`. These objects are created by the constructor call `Vector2(x,y)`. Assume that you have the

When dealing with objects, you should remember that a variable does not *contain* an object. It can only hold the *identifier* of the object. Therefore, the assignment statements

```
   u = Vector2(1.0,2.0)
```

```
    v = Vector2(3.0,4.0)
```

would produce the following contents in both global space and the heap:



In this case we have folders representing two `Vector2` objects, and two global variables storing these object identifiers. We made up the folder identifier numbers, as their value does not matter. If we ask you to make a `Vector2` object, you should feel free to make up the identifier number, but make sure it is different from the ones currently in `u` and `v`.

Part C: Diagram the Execution of `perp(u)`

The function `perp` takes a `Vector2` object as an argument, and modifies it so that it is perpendicular to the original vector. If you are not familiar with how vectors work, that is not a problem. All you need to know is the following function definition:

```python
def perp(v):
    """Modifies v to be perpendicular to the original

    Precondition: v is a Vector2 object"""
    tmp = v.y
    v.y = v.x
    v.x = -tmp
```

The function `perp` is a procedure (it has no return statement) and so it does not need to be used in an assignment statement. Hence the call

```python
    perp(u)
```

is a statement by itself.

Diagram this call given the global variables and heap contents (i.e the folders) shown above. Your answer will in include a sequence of diagrams for the call frame for `perp(u)`. Again, you should have a diagram when the frame is created, a diagram for each line executed, and a diagram for deleting the frame.

You should draw global space and the heap next to every single diagram in your evolution of the frame. If there are no changes to these areas of memory, you are allowed to write UNCHANGED. However, if the contents of any global variable change, or the contents of the object folders are altered, you should cross out the old value and write in the new value. Again, you only need to draw the old values at the step the change takes place.

Part D: Diagram the Execution of `v = perp2(u)`

The function `perp2` is similar to `perp`, except that it is a fruitful function instead of a procedure. We define the function as follows:

```
Import vect

def perp2(v):
    """Returns a vector perpendicular to v

    Precondition: v is a Vector2 object"""
    u = vect.Vector2(0.0,0.0)
    u.y = v.x
    u.x = -v.y
    return u
```

Diagram the execution of the assignment statement

```
   v = perp2(u)
```

given the contents of global space and the heap shown above. Assume that you are starting over with `u` and `v`; **do not include any changes from Part C**.

As always, you should have a diagram when the frame is created, a diagram for each line executed, and a diagram for deleting the frame. You should draw the global space and heap space next to every single diagram in your evolution of the frame. If the contents of any global variable change, or the contents of the object folders are altered, you should cross out the old value and write in the new value (for the step at which they are altered).

Note: Technically the call to the function (constructor) `vect.Vector2` should create a second call frame. However, you do not know the definition of that function, and so we do not want you to draw its frame.

Working in Reverse

So far in this assignment, we have given you a function definition and asked you to draw the diagram for a function call. Now we want you to do the reverse. We give you the diagrams for several function calls and ask you to write the corresponding definition. All you have to do is show the Python code. Any function definition is correct **so long as it produces the frame shown for each call**.

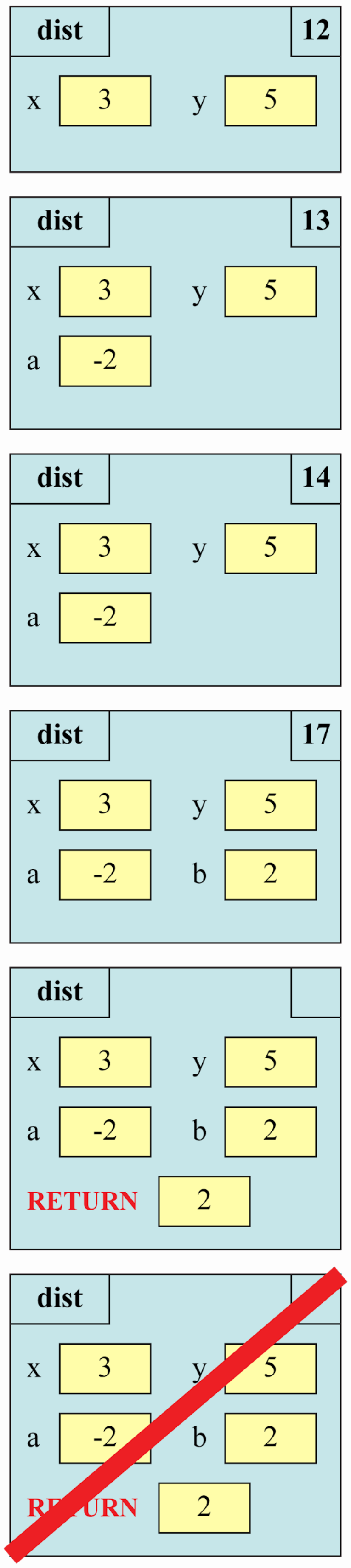
Part E: Define the function `dist(x,y)`

The function `dist` has the following specification:

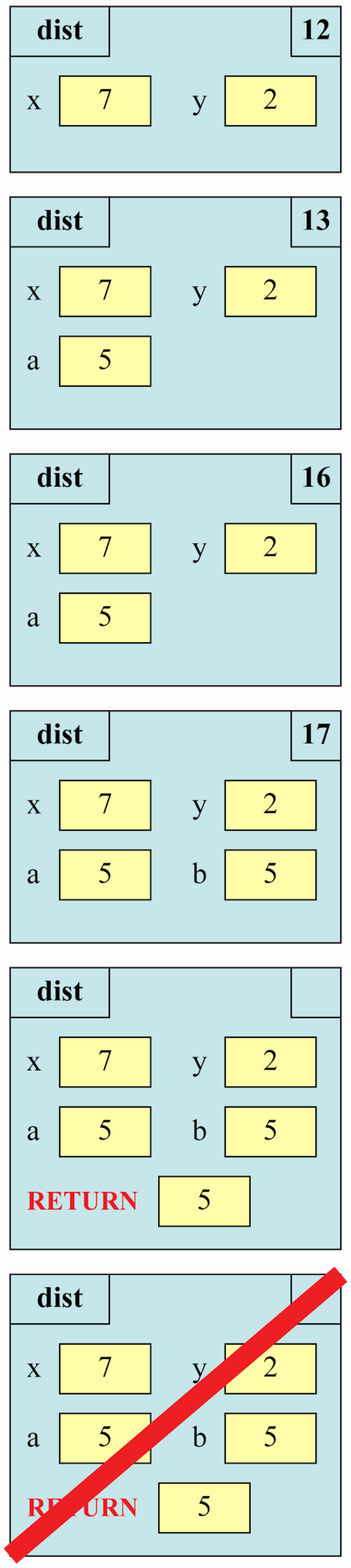
```
def dist(x,y):
    """Returns: The number line distance between x and y

    Example: dist(2,5) returns 3
    Example: dist(5,2) returns 3

    Parameter x: the starting point
    Precondition: x is a number

    Parameter y: the ending point
    Precondition: y is a number"""
```

The following diagrams illustrate the function calls `dist(3,5)` and `dist(7,2)`, respectively.

Call: dist(3,5)



Call: dist(7,2)



Complete the function definition.

Hint: Your solution cannot use the built-in function `abs`, as that would technically create a call frame for that function as well.

Finishing Touches

The assignment should carry the names of all the group members.

Creating a PDF for Submission

We want you to submit your work as a PDF. If you created your assignment on a computer, this should be relatively easy. Otherwise, you will need to scan your paper as a PDF. Your solution should be a single PDF file.

Uploading the PDF

Upload your pdf files here:
https://drive.google.com/drive/folders/1m7x5fxyJbhUpRbR-KaZhg-sHjpgSDyFJ?usp=share_link