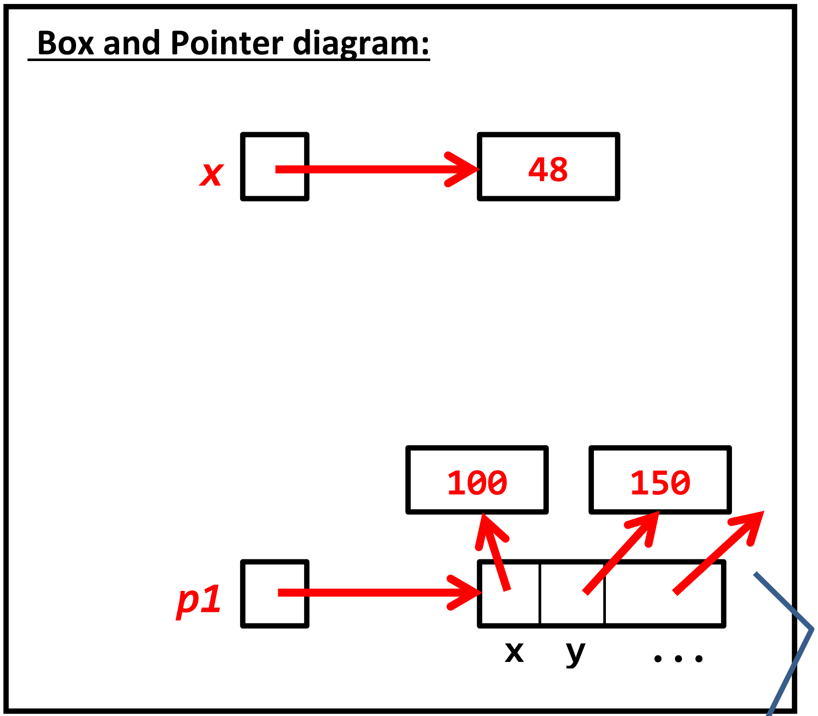
Name: ______________________________

1. With your instructor, draw a Box and Pointer diagram that shows what happens when the following statements execute. (Use the boxes we supplied; just add labels and arrows for variables and data for objects.)

```
x = 48
p1 = zg.Point(100, 150)
```

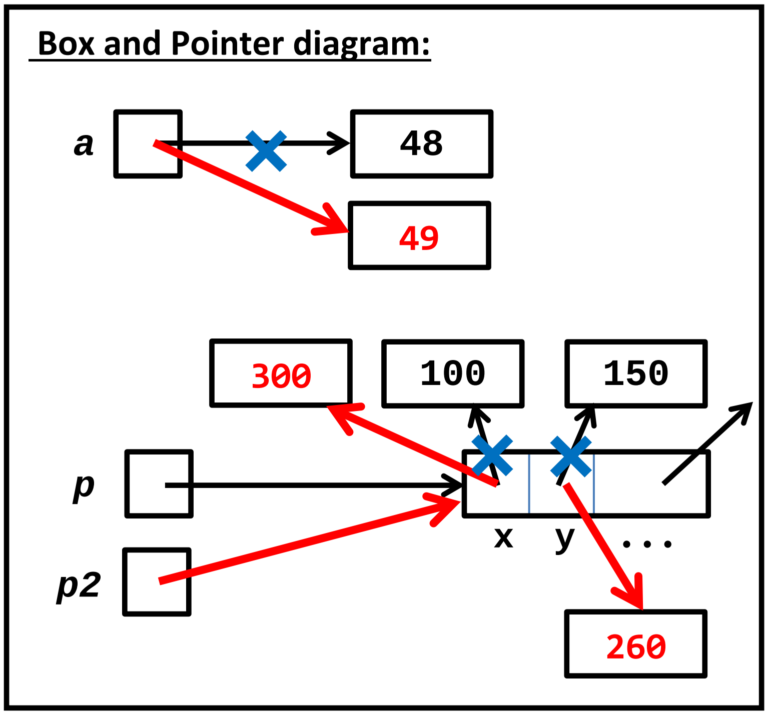


The 3rd and other arrows point to the Point's color, etc.

2. An ***assignment statement*** causes an arrow to be *established* or *changed*. That's true for fields as well as ordinary variables. ***The arrows always point to objects, never to other variables.***

With your instructor, draw a Box and Pointer diagram that shows what happens when the statements below execute. (We've already done the first two statements.)

```
a = 48
p = zg.Point(100, 150)

a = a + 1
p.x = 300
p2 = p
p2.y = p.x - 40
```



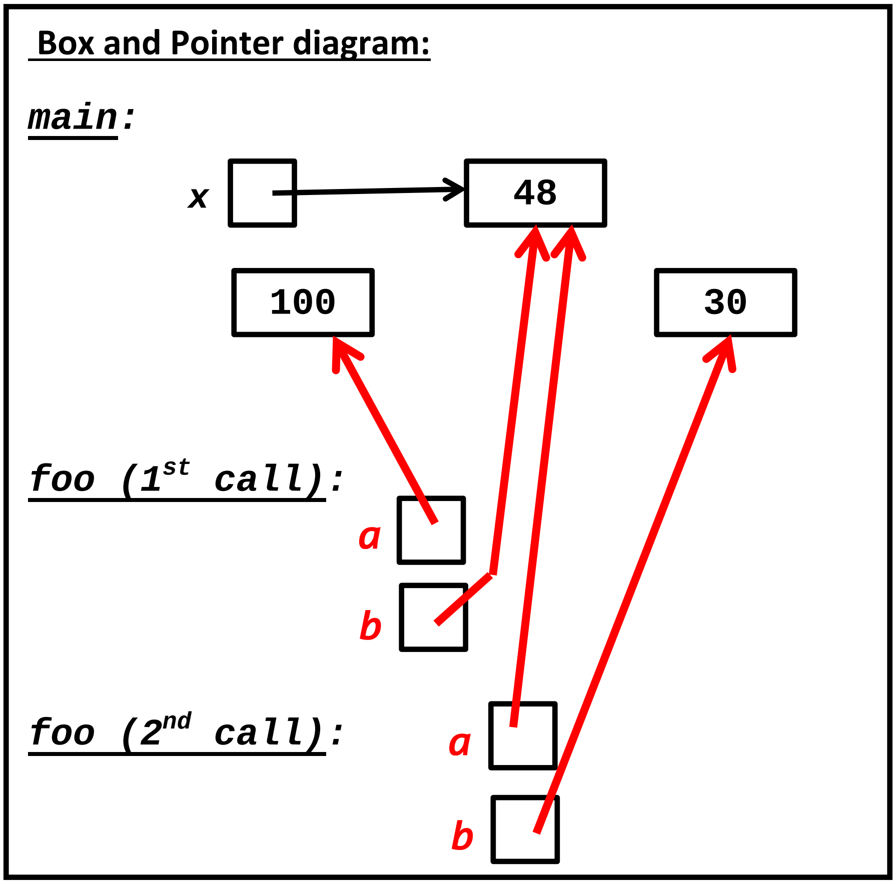
In doing this exercise, note that it is perfectly OK to have two variables refer to the ***same*** object.

3. A ***function call*** creates a new ***namespace*** in which the function will run. The ***parameters*** are variables in that namespace. When the function is called, the first thing that happens is that each parameter is assigned the ***value*** of the corresponding actual argument.

   For example, in the code snippet below when `foo(100, x)` executes, the parameter `a` is assigned the value `100`, just as if the statement `a = 100` were executed.

   With your instructor, draw a Box and Pointer diagram that shows what happens when *main* (below) executes.

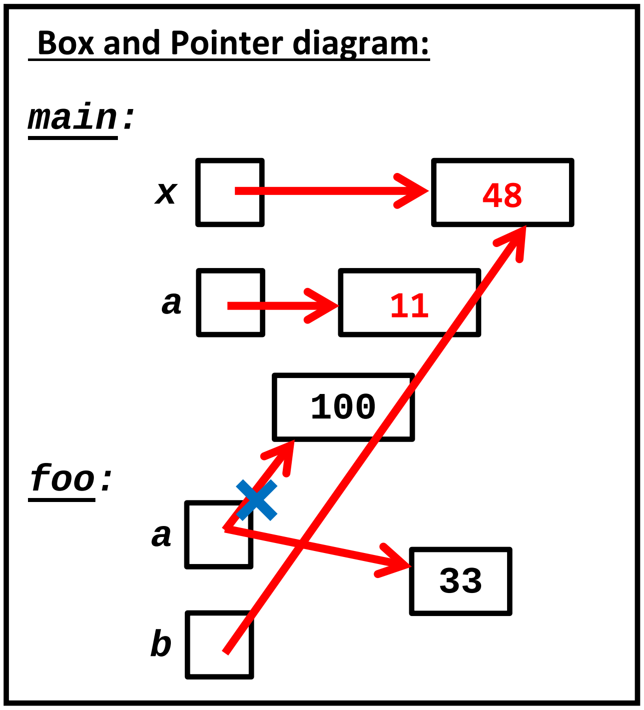
```
def main():
    x = 48
    foo(100, x)
    foo(x, 30)

def foo(a, b):
    ...
```



4. As you saw in the previous problem, each ***function call*** creates a new ***namespace*** in which the function will run. Variables in that namespace are simply *not the same* as variables with the same name in *main* or other namespaces. Try this one:

Complete the Box-and-Pointer diagram to the right to show what happens when *main* (below) executes. Also show the output that is printed.

```
def main():
    x = 48
    a = 11
    foo(100, x)
    print('C.', x, a)


def foo(a, b):
    print('A.', a)
    a = 33
    print('B.', a)
```



Output:

A. ____100_____

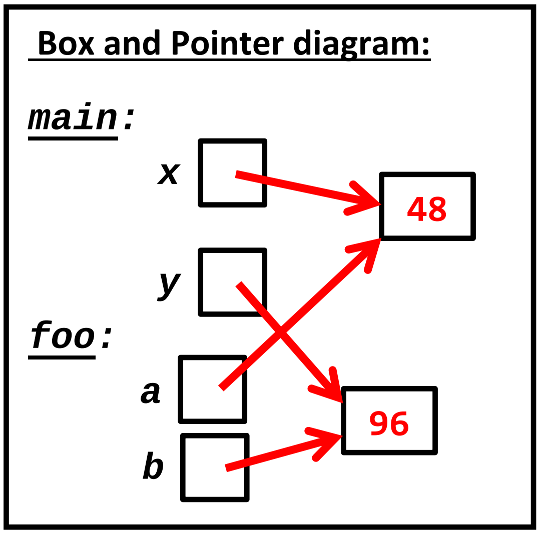
B. ____33_________

C. ___48___11___

5. As you know, you can send information "back" from a function to its caller by using a ***return*** statement. Try this one to see how that appears in these diagrams:

Complete the Box-and-Pointer diagram to the right to show what happens when *main* (to the right) executes.

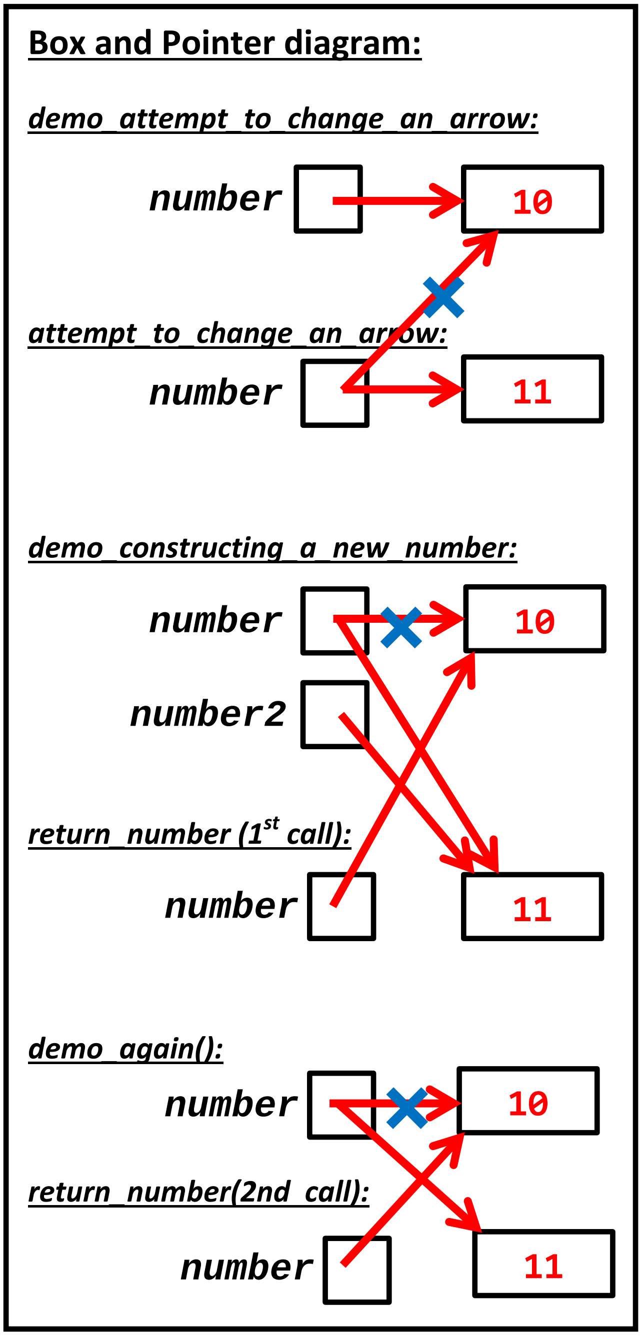
```
def main():
    x = 48
    y = foo(x)


def foo(a):
    b = 2 * a
    return b
```



6. It is simply not possible for a function to change the arrow in the ***caller*** that corresponds to one of the function's arguments. If you really want to accomplish something like that, you have to return a value and re-assign the variable that points to the argument to that returned value. Try this one to see those ideas in action:

   Complete the Box-and-Pointer diagram to the right to show what happens when *main* (below) executes. Also show the output that is printed.

```
def main():
    demo_attempt_to_change_an_arrow()
    demo_constructing_a_new_number()
    demo_again()

def demo_attempt_to_change_an_arrow():
    number = 10
    attempt_to_change_an_arrow(number)
    print('B.', number)

def attempt_to_change_an_arrow(number):
    number = number + 1
    print('A.', number)

def demo_constructing_a_new_number():
    number = 10
    number2 = return_number(number)
    print('C.', number, number2)
    number = number2
    print('D.', number, number2)

def demo_again():
    number = 10
    number = return_number(number)
    print('E.', number)

def return_number(number):
    return (number + 1)
```



Output: A. _____11______

B.________10__________

C._____10___11______

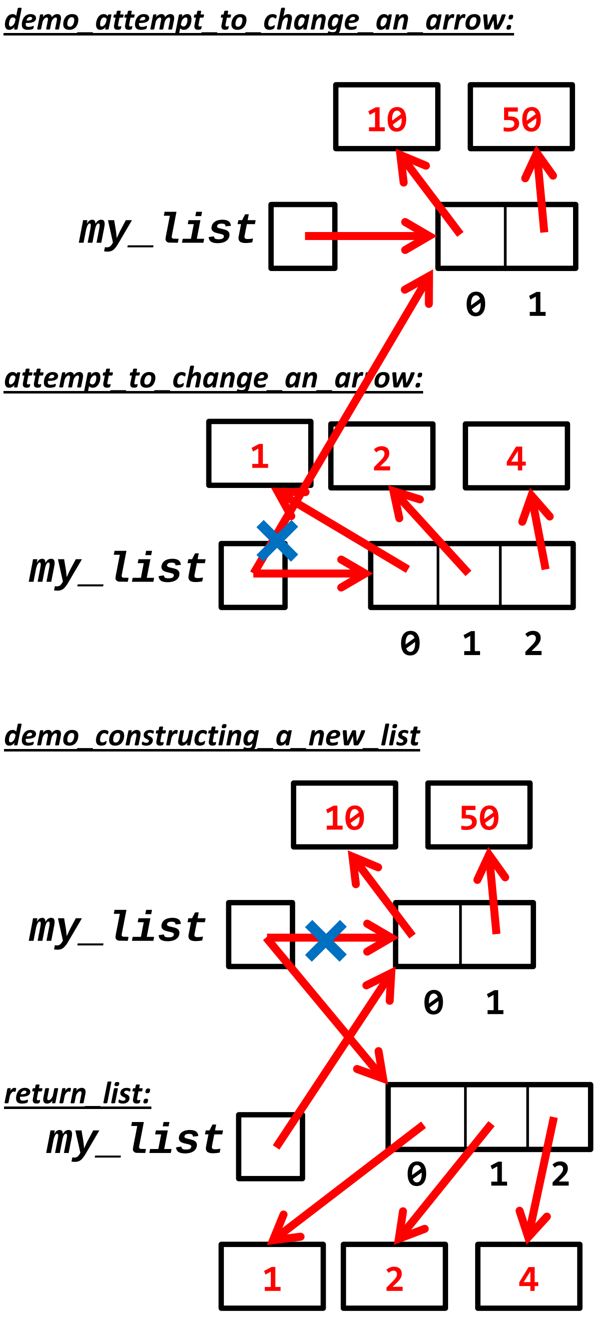
D._____11___11_________

E._______11_______________

7. There is nothing special about using *numbers* in the preceding exercise. To see this, draw a Box-and-Pointer diagram that shows what happens when *main* (below) executes. Also show the output that is printed. (This example is similar to the previous one, but with *lists* instead of numbers.)

```
def main():
    demo_attempt_to_change_an_arrow()
    demo_constructing_a_new_list()

def demo_attempt_to_change_an_arrow():
    my_list = [10, 50]
    attempt_to_change_an_arrow(my_list)
    print('B.', my_list)

def attempt_to_change_an_arrow(my_list):
    my_list = [1, 2, 4]
    print('A.', my_list)

def demo_constructing_a_new_list():
    my_list = [10, 50]
    my_list = return_list(my_list)
    print('C.', my_list)

def return_list(my_list):
    return [1, 2, 4]
```

Box and Pointer diagram:



Output:

A. [1, 2, 4]

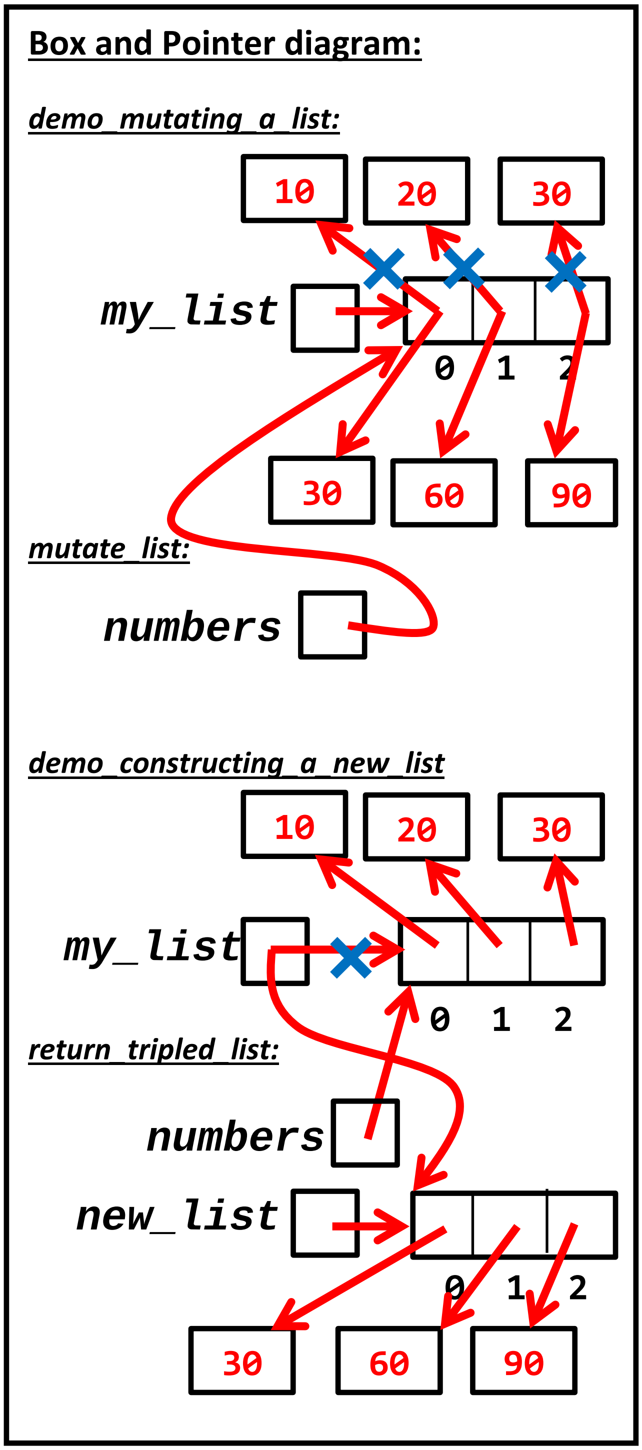
B. [10, 50]

C. [1, 2, 4]

8. We have seen that it is simply not possible for a function to change the arrow in the ***caller*** that corresponds to one of the function's arguments. But many objects can be ***mutated***, which means that ***the object's value (not the variable's reference) changes.***

   To see this, draw a Box and Pointer diagram that shows what happens when *main* (below) executes. Also show the output that is printed. Do NOT show boxes for the loop variables ***k*** and ***number***, since that would clutter the diagram.

```
def main():
    demo_mutating_a_list()
    demo_constructing_a_new_list()

def demo_mutating_a_list():
    my_list = [10, 20, 30]
    mutate_list(my_list)
    print('A.', my_list)

def mutate_list(numbers):
    for k in range(len(numbers)):
        numbers[k] = numbers[k] * 3

def demo_constructing_a_new_list():
    my_list = [10, 20, 30]
    my_list = return_tripled_list(my_list)
    print('B.', my_list)

def return_tripled_list(numbers):
    new_list = []
    for number in numbers:
        new_list.append(number * 3)

    return new_list
```



Output:

A. [30, 60, 90]

B. [30, 60, 90]

mutate_list and ***return_tripled_list*** both end up with a tripled list. ***Which one uses less storage?*** [✓ *mutate_list*] ***return_tripled_list*** (circle your choice)

9. As you just saw, ***lists are mutable*** – the value of the object itself (that is, its "insides") can change.

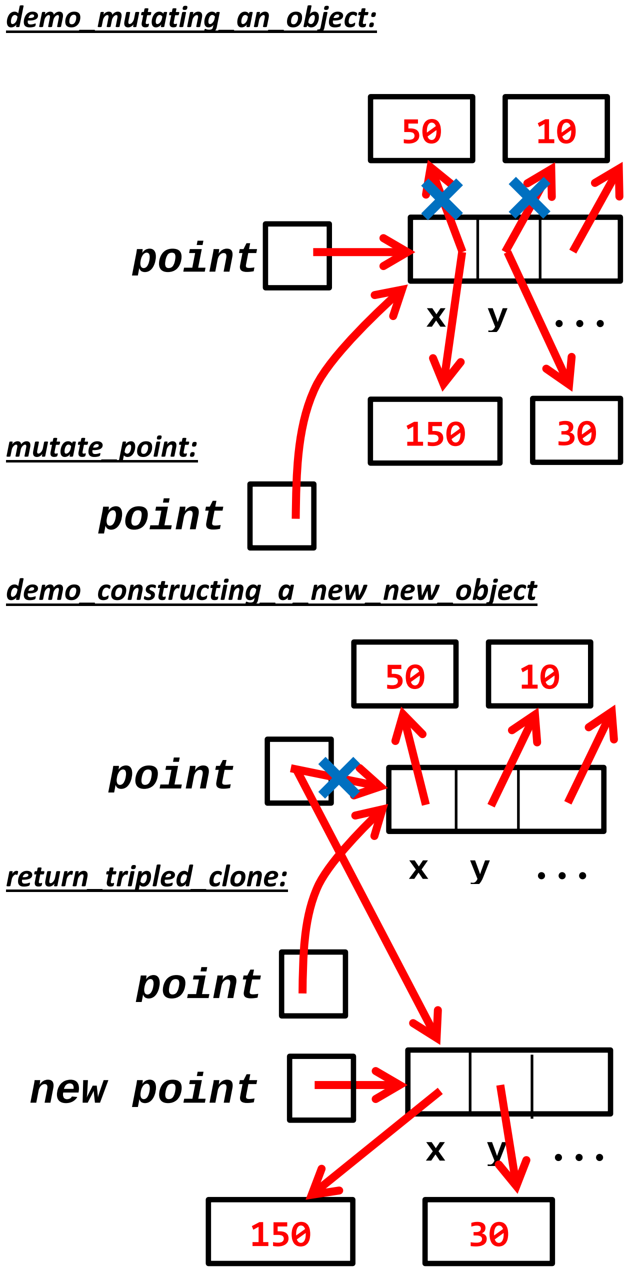
   Tuples are ***NOT mutable*** – that is their primary difference from lists. ***Strings*** are ***NOT mutable*** and ***numbers*** are ***NOT mutable***.

   Instances of user-defined classes (like the Zellegraphics objects) ***are, in general, mutable***.

   To see this, draw a Box and Pointer diagram that shows what happens when *main* (below) executes. Also show the output that is printed.

```
def main():
    demo_mutating_an_object()
    demo_constructing_a_new_object()

def demo_mutating_an_object():
    point = zg.Point(50, 10)
    mutate_point(point)
    print('A.', point)

def mutate_point(point):
    point.x = point.x * 3
    point.y = point.y * 3

def demo_constructing_a_new_object():
    point = zg.Point(50, 10)
    point = return_tripled_clone(point)
    print('B.', point)

def return_tripled_clone(point):
    new_point = zg.Point(point.x * 3,
                         point.y * 3)
    return new_point
```

Box and Pointer diagram:



Output:

A. Point(150, 30)

B. Point(150, 30)

`mutate_point` and *`return_tripled_clone`* both end up with a tripled point. ***Which one uses less storage?*** [✓ *`mutate_point`*] *`return_tripled_clone`* (circle your choice)