

Exam 2 – Practice Problems for the Paper-and-Pencil portion

SOLUTION (check these against your answers page by page after attempting each page)

Exam 2 will assess material covered in Sessions 1 through 14. The paper-and-pencil part will draw problems especially from the following concepts. The numbers in the brackets at the beginning of the item are problems that let you practice that concept.

Problems may include **tracing** or **writing** code (as well as questions of other forms, like True/False and multiple choice).

- [1, 2, 4, 23, 37] **Scope**, especially scope *inside a method*.

- [1, 4, 5, 37] **Flow of control** through **function calls** and **returns**. Including calls within expressions, e.g.

```
print(foo1(...), foo2(...)) or  
z = foo1(...) + foo2(...) + foo1(...) or  
w = foo1( foo3(...) + foo4(...) + ... )
```

Especially problems that assess the above by **tracing code by hand**.

- [3, 6, 19, 20 - 22] **Range expressions**: All 3 forms.

- [5, 7 - 12, 17, 18, 33, 36] **Indexing into a sequence**, especially for the 1st and last items in the sequence. **Lists, tuples** and **strings**. Out of bounds errors, including (failed) attempts to accumulate by statements like this **WRONG code**:

```
s = []  
for k in range(...):  
    x[k] = ...
```

- [19, 20 – 22, 42] Code that **loops** using a **range** statement and **sums, counts**, or **accesses items in a sequence**. Especially problems that assess the above by **tracing code by hand**.
- [13 - 18, 21] **Concatenating** items to a sequence. Doing so by using the **+ operator** for strings and tuples and the **append** method for lists.

- [19 - 22] **Writing simple functions** that **loop through a sequence** and:
 - **access** (e.g. sum/count), doing **all or part** of the sequence, **forwards or backwards**
 - **find**
 - **accumulate**
 - get **max or min**
 - access **two places in the sequence** in each iteration of the loop
 - access two sequences **in parallel**
 as well as combinations of the above.
- [13 - 16, 24 - 32, 38, 39] **References** and related concepts.
- [30 - 32, 38, 39] **Box and pointer diagrams**.
- [27 - 29, 33 - 35] **Mutation** of a list / object by a function.
- [10 - 12, 34, 35] Fact that tuples and strings are **immutable**. What that means.
- [4, 37, 40, 41] Implementing a class. What **SELF** is. How to use it.
- [4, 37] **Tracing by hand** code that includes **method calls** on objects from a class whose code is given.

The actual exam's paper-and-pencil part will be much shorter than this collection of practice problems. That said, all of these practice problems are excellent practice for Exam 2.

Pay special attention to Problems:

3 4 5 19 – 22 24 – 26 29 – 32 and 37 – 42

since they summarize many of the concepts and are in forms that we often use for exams.

Also be sure to review your Exam 1 paper-and-pencil problems, since you may see similar questions on some of those concepts.

1. Consider the code snippets defined below. They are contrived examples with poor style but will run without errors. For each, what does it print when *main* runs? (Each is an independent problem. Pay close attention to the order in which the statements are executed.)

```
def main():
    x = 5
    y = 3
    print('main 1', x, y)
    foo(x, y)
    print('main 2', x, y)

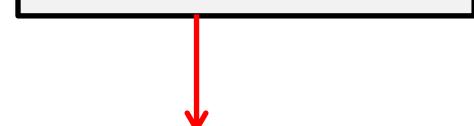
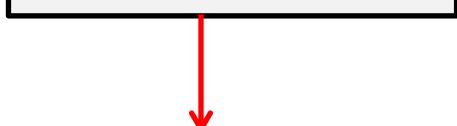
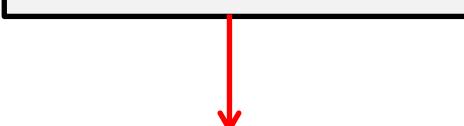
def foo(a, b):
    print('foo 1', a, b)
    a = 66
    b = 77
    x = 88
    y = 99
    print('foo 2', a, b,
          x, y)
```

```
def main():
    x = 5
    y = 3
    print('main 1', x, y)
    foo(x, y)
    print('main 2', x, y)

def foo(x, y):
    print('foo 1', x, y)
    a = 66
    b = 77
    x = 88
    y = 99
    print('foo 2', a, b,
          x, y)
```

```
def main():
    x = 5
    y = 3
    print('main 1', x, y)
    foo(y, x)
    print('main 2', x, y)

def foo(x, y):
    print('foo 1', x, y)
    a = 66
    b = 77
    x = 88
    y = 99
    print('foo 2', a, b,
          x, y)
```



Prints: main 1 5 3

foo 1 5 3

foo 2 66 77 88 99

main 2 5 3

Prints: main 1 5 3

foo 1 5 3

foo 2 66 77 88 99

main 2 5 3

Prints: 1 5 3

foo 1 3 5

foo 2 66 77 88 99

main 2 5 3

2. Consider the code snippet to the right. Both ***print*** statements are wrong.

- Explain why the first ***print*** statement (in ***main***) is wrong.

The name *z* in *main* is not defined.

(The *z* in ***foo*** has nothing to do with the *z* in ***main***.)

- Explain why the second ***print*** statement (in ***foo***) is wrong.

The name *x* in *foo* is not defined.

(The *x* in ***main*** has nothing to do with the *x* in ***foo***.)

```
def main():
    x = 5
    foo(x)
    print(z)
```

```
def foo(a):
    print(x)
    z = 100
    return z
```

3. Consider the code snippet below. It is a contrived example with poor style, but it will run without errors. What does it print when it runs?

Write your answer in the box to the right.

```
b = [44]
a = (50, 30, 60, 77)
x = 3

for k in range(len(a)):
    b = b + [a[x - k]]
    print(k, b)

print('A.', a)
print('B.', b)
print('X.', x)
```

Output: (I have included some extra spaces to make the answer easier to read.)

0 [44, 77]

1 [44, 77, 60]

2 [44, 77, 60, 30]

3 [44, 77, 60, 30, 50]

A. (50, 30, 60, 77)

B. [44, 77, 60, 30, 50]

X. 3

4. Consider the code on the page to the right of this page. It is a contrived example with poor style but will run without errors. In this problem, you will trace the execution of the code. As each location is encountered during the run, in the table below:

- **CIRCLE** each variable that is **defined** at that location.
- **WRITE** the **VALUE** of each variable that you **circled** directly **BELLOW** the circle.

For example, the run defines the functions and then calls **main**, as usual. The first of the eleven locations **to be encountered** is **Location 8**. At Location 8, the only variable defined is **a**, with value **44** at that point of the program's run. So, on the row for Location 8, you would circle **a** and write **44** directly below it.

Note that you fill out the table **in the order that the locations are encountered, NOT from top to bottom. ASK FOR HELP IF YOU DO NOT UNDERSTAND WHAT THIS PROBLEM ASKS YOU TO DO.**

When Location 1 is encountered the 1 st time	a 10	m	m1.a	m1.m	m2.a	m2.m	self.a	self.m
When Location 2 is encountered the 1 st time	a 10	m	m1.a	m1.m	m2.a	m2.m	self.a 3	self.m 15
When Location 1 is encountered the 2 nd time	a 22	m	m1.a	m1.m	m2.a	m2.m	self.a	self.m
When Location 2 is encountered the 2 nd time	a 22	m	m1.a	m1.m	m2.a	m2.m	self.a 3	self.m 27
Location 3	a	m	m1.a	m1.m	m2.a	m2.m	self.a 3	self.m 42
Location 4	a	m	m1.a	m1.m	m2.a	m2.m	self.a 103	self.m 42
Location 5	a 44	m	m1.a	m1.m	m2.a	m2.m	self.a	self.m
Location 6	a 6	m 31	m1.a	m1.m	m2.a	m2.m	self.a	self.m
Location 7	a 6	m 31	m1.a 3	m1.m 42	m2.a 3	m2.m 27	self.a	self.m
Location 8	a 44	m	m1.a	m1.m	m2.a	m2.m	self.a	self.m
Location 9	a 44	m	m1.a	m1.m	m2.a 3	m2.m 42	self.a	self.m
Location 10	a 44	m	m1.a	m1.m	m2.a 103	m2.m 42	self.a	self.m
Location 11	a	m	m1.a	m1.m	m2.a	m2.m	self.a	self.m

Showing your work (by marking up the code, drawing a box-and-pointer diagram, or any other way you wish) is the best way to allow for partial credit.

Feel free to use a separate blank sheet of paper if you like.

```

class Mini(object):
    def __init__(self, a):
        ##### Location 1
        self.a = 3
        self.m = a + 5
        ##### Location 2

    def blah(self):
        ##### Location 3
        self.a = self.a + 100
        ##### Location 4

def foo(a):
    ##### Location 5
    a = 6
    m = 31
    ##### Location 6
    m1 = Mini(10)
    m2 = Mini(22)
    m1.m = m1.m + m2.m
    # Location 7
    return m1

def main():
    a = 44
    ##### Location 8
    m2 = foo(a)
    ##### Location 9
    m2.blah()
    ##### Location 10

main()
##### Location 11

```

**ASK FOR HELP IF YOU DO NOT UNDERSTAND
WHAT THIS PROBLEM ASKS YOU TO DO.**

The locations are encountered in the following order:

1. Location 8
2. Location 5
3. Location 6
4. Location 1 the 1st time
5. Location 2 the 1st time
6. Location 1 the 2nd time
7. Location 2 the 2nd time
8. Location 7
9. Location 9
10. Location 3
11. Location 4
12. Location 10
13. Location 11

5. Consider the code snippet below. It is a contrived example with poor style, but it will run without errors. What does it print when it runs?

Write your answer in the box to the right of the code.

```

def main():
    a = alpha(3)

    print()
    b = beta(2)

    print()
    g = gamma(6)

    print()
    c = alpha(beta(10))

    print()
    print("main!", a, b, g, c)

def alpha(x):
    print("Alpha!")
    return x + 7

def beta(y):
    print("Beta!")
    return 5 + alpha(y + 10)

def gamma(z):
    print("Gamma!", alpha(1), beta(1))
    return (alpha(z) + beta(z - 3)
           + alpha(z + 2))

main()

```

Output: (I have included some extra spaces to make the answer easier to read.)

Alpha!

Beta!

Alpha!

Alpha!

Beta!

Alpha!

Gamma! 8 23

<no space really here>

Alpha!

Beta!

Alpha!

Alpha!

Beta!

Alpha!

Alpha!

main! 10 24 53 39

6. For each of the following **range** expressions, write the sequence that it generates. Write **empty** if the generated sequence is the empty sequence (i.e., has no items in it). We have done the first two for you as examples.

- `range(6)` generates the sequence: **0 1 2 3 4 5**
- `range(6, 6)` generates the sequence: **empty (nothing, the empty range)**
- `range(3, 6)` generates the sequence: **3 4 5**
- `range(12, 6)` generates the sequence: **empty (nothing, the empty range)**
- `range(3, 8, 1)` generates the sequence: **3 4 5 6 7**
- `range(3, 8, 2)` generates the sequence: **3 5 7**
- `range(4, 8, 2)` generates the sequence: **4 6**
- `range(5, 14, 3)` generates the sequence: **5 8 11**
- `range(5, 15, 3)` generates the sequence: **5 8 11 14**
- `range(20, 15, -1)` generates the sequence: **20 19 18 17 16**
- `range(20, 15)` generates the sequence: **empty (nothing, the empty range)**
- `range(15, 20, -1)` generates the sequence: **empty (nothing, the empty range)**
- `range(20, 17, -3)` generates the sequence: **20**
- `range(20, 16, -3)` generates the sequence: **20 17**
- `range(20, 20, -3)` generates the sequence: **empty (nothing, the empty range)**
- `range(5, 0, -1)` generates the sequence: **5 4 3 2 1**
- `range(5, -1, -1)` generates the sequence: **5 4 3 2 1 0**
- `range(5, -1, -3)` generates the sequence: **5 2**
- `range(5, -2, -3)` generates the sequence: **5 2 -1**
- `range(8)` generates the sequence: **0 1 2 3 4 5 6 7**
- `range(100, 100)` generates the sequence: **empty (nothing, the empty range)**

7. Consider the list `X = [3, 7, 1, 0, 99, 5]`.

For each of the following print statements, indicate what would be printed. Write ERROR if the print statement would generate an exception (error).

- `print(len(X))` would print: **6**
- `print(X[0])` would print: **3**
- `print(X[1])` would print: **7**
- `print(X[5])` would print: **5**
- `print(X[6])` would print: **ERROR**
- `print(X[-1])` would print: **5**
- `print(X[-6])` would print: **3**
- `print(X[-7])` would print: **ERROR**
- `print(X[len(X)])` would print: **ERROR**
- `print(X[len(X) - 1])` would print: **5**

8. Consider the tuple `T = (4, 10, 3)`.

For each of the following print statements, indicate what would be printed. Write ERROR if the print statement would generate an exception (error).

- `print(len(T))` would print: **3**
- `print(T[0])` would print: **4**
- `print(T[2])` would print: **3**
- `print(T[3])` would print: **ERROR**
- `print(T[-1])` would print: **3**
- `print(T[len(T)])` would print: **ERROR**
- `print(T[len(T) - 1])` would print: **3**

9. Consider the string `s = 'hello'`.

For each of the following print statements, indicate what would be printed. Write ERROR if the print statement would generate an exception (error).

- `print(len(s))` would print: **5**
- `print(s[0])` would print: **h**
- `print(s[4])` would print: **o**
- `print(s[5])` would print: **ERROR**
- `print(s[len(s)])` would print: **ERROR**
- `print(s[len(s) - 1])` would print: **o**
- `print(s[-1])` would print: **o**

10. Consider the list `X = [3, 7, 1, 0, 99, 5]` and the statement:

`X[3] = 100`

Would the above statement would generate an exception (error)? **Yes** or **No** (circle your answer)

11. Consider the tuple `T = (3, 7, 1, 0, 99, 5)` and the statement:

`T[3] = 100`

Would the above statement would generate an exception (error)? **Yes** or **No** (circle your answer)

12. Consider the string `s = 'hello'` and the statement:

`s[3] = 'y'`

Would the above statement would generate an exception (error)? **Yes** or **No** (circle your answer)

13. Consider a tuple `T`. Write a statement that would make `T` refer to a new tuple with the same items as it currently has, but also with `74` appended to the end of `T`.

`T = T + (74,)` **(Note the comma!)**

14. Consider a string `s`. Write a statement that would make `s` refer to a new string with the same characters as it currently has, but also with '`r`' appended to the end of `s`.

`s = s + 'r'` **or** `s = s + "r"`

15. Consider a list `X`. Write a statement that would make `X` refer to a **new** list with the same items as it currently has, but also with '`r`' appended to the end of `X`.

`X = X + ['r']` **or** `X = X + ["r"]` **(Note the square brackets!)**

16. Consider a list `X`. Write a statement that would make `X` refer to the **same list**, but with that list having had '`r`' appended to the end of `X`.

`X.append('r')` **or** `X.append("r")` **(Note the ABSENCE of square brackets!)**

Note: The **append** approach is more efficient (in its use of time and space) than the **+ operator** approach. Hence, use the **append** approach for concatenating to a **list** unless directed otherwise.

17. Consider the following code snippet:

```
X = []
X[0] = 100
X[1] = 77
X[2] = 88
```

Would the above statements generate an exception (error)? **Yes** or **No** (circle your answer)

18. Consider the following code snippet:

```
X = []
X.append(100)
X.append(77)
X.append(88)
```

Would the above statements generate an exception (error)? **Yes** or **No** (circle your answer)

If not, what would be the **value of the list X** after the above statements execute?

[100, 77, 88]

19. Consider a sequence named **X**. Write statements that would:

- Print the first (beginning) item of the sequence:

print(X[0])

- Print the last item of the sequence:

print(X[len(X) - 1]) or print(X[-1])

- Print all the items of the sequence, one by one, from beginning to end:

Note: in this and the following sub-problems of this problem, the statement inside the loop should be something like **print(X[k])**. That is, let the **range** statement do all the heavy lifting.

for k in range(len(X)):
 print(X[k])

or

for item in X:
 print(item)

Note: the latter form is more elegant and hence the choice of an experienced Python software developer. However, there are problems in which one needs the index (here, *k*) in the body of the loop; those problems require the first form above. Either form is acceptable for our course.

- Print all the items of the sequence, one by one, from **end** to beginning:

```
for k in range(len(X) - 1, -1, -1):
    print(X[k])
```

or

```
for k in range(-1, -len(X) - 1, -1):
    print(X[k])
```

There are other alternatives as well that put the heavy lifting into the argument for the index inside the body of the loop, e.g.:

```
for k in range(len(X)):
    print(X[len(X) - k - 1])
```

- Print all the items at **odd indices** of the sequence, one by one, beginning to end:

```
for k in range(1, len(X), 2):
    print(X[k])
```

There are other alternatives as well, but following one is **inferior** because it takes roughly twice as long to run as the above solution:

```
for k in range(len(X)):
    if k % 2 == 1:
        print(X[k])
```

- Print all the items of the sequence that are **odd**, one by one, beginning to end (for this problem, assume that the sequence contains only positive integers):

```
for k in range(len(X)):
    if X[k] % 2 == 1:
        print(X[k])
```

Notice how different this problem is from the previous (but similar-sounding) one!

- Starting at the **second-to-last** item in the sequence and going **backwards**, print **every 4th item** in the sequence:

```
for k in range(len(X) - 2, -1, -4):
    print(X[k])
```

or **for k in range(-2, -len(X) - 1, -4):**
print(x[k])

20. Write a function (including its `def` line) named `count_small` that takes a sequence of numbers and a number `Z`, and returns the number of items in the sequence that are less than `Z`. For example:

`count_small([8, 2, 7, 10, 20, 1], 7)` returns `2` (since 2 and 1 are less than 7)

`count_small([8, 2, 7, 10, 20, 1], -4)` returns `0`

```
def count_small(sequence, z):
    count = 0
    for k in range(len(sequence)):
        if sequence[k] < z:
            count = count + 1
    return count
```

or

```
def count_small(sequence, z):
    count = 0
    for number in sequence:
        if number < z:
            count = count + 1
    return count
```

21. Write a function (including its `def` line) named `get_all_at_even_indices` that takes a sequence and returns a list of the items in the sequence at even-numbered indices. For example:

`get_all_at_even_indices([8, 2, 7, 10, 20])` returns `[8, 7, 20]`

`get_all_at_even_indices('abcdefg')` returns `['a', 'c', 'e', 'g']`

```
def get_all_at_even_indices(sequence):
    items = []
    for k in range(0, len(sequence), 2):
        items.append(sequence[k])
    return items
```

The statement in the above:

`items.append(sequence[k])`

could have been written as:

`items = items + [sequence[k]]`

but that would be a less efficient solution. The `append` method applies only to *lists*, so if a problem requires building a *string* or *tuple*, you would normally use the `+ operator` approach. (Or, better if the number of items is large, convert the string/tuple to a list, use `append` to build the list, then convert the list to the required string/tuple.)

22. Write a function (including its `def` line) named `get_first_even_x` that takes a sequence of `rg.Circle` objects and returns the radius of the first `rg.Circle` in the sequence whose center's x-coordinate is even, or `-999` if there are no such circles in the sequence. For example:

```
get_first_even_x ([rg.Circle(rg.Point(115, 20), 50),
                   rg.Circle(rg.Point(8, 1), 33),
                   rg.Circle(rg.Point(12, 2), 22)])      returns 33

get_first_even_x ([rg.Circle(rg.Point(115, 20), 50),
                   rg.Circle(rg.Point(37, 22), 33),
                   rg.Circle(rg.Point(11, 2), 22)])      returns -999
```

```
def get_first_even_x(circles):
    for k in range(len(circles)):
        circle = circles[k]
        if circle.center.x % 2 == 0:
            return circle.radius
    return -999
```

Here is a nice alternative:

```
def get_first_even_x(circles):
    for circle in circles:
        if circle.center.x % 2 == 0:
            return circle.radius
    return -999
```

Note how I chose to give a name to `circles[k]`, that is, to call it `circle` (the singular form of the sequence named `circles`.) Consider doing so routinely, since it makes the following code be more familiar (no square brackets).

CRITICAL: Note that the `return -999` is AFTER the loop, not inside the loop!

23. Consider the following two candidate function definitions:

```
def foo():
    print('hello')
```

```
def foo(x):
    print(x)
```

- Which is “better”? Circle the better function.
- Briefly explain why you circled the one you did.

The form to the right allows the caller of the function to print ANYTHING, while the form to the left is useful only for printing 'hello'. In general, **adding parameters** to a function makes the function **more powerful**.

24. True or false: **Variables are REFERENCES to objects.** True False (circle your choice)

25. True or false: **Assignment** (e.g. `x = 100`) causes a variable to refer to an object. True False (circle your choice)

26. True or false: **Function calls** (e.g. `foo(54, x)`) also cause variables to refer to objects. True False (circle your choice)

27. Give one example of an object that is a **container** object:

Here are several examples: a **list**, a **tuple**, an **rg.Circle**, a **Point**, an **rg.RoseWindow**. In general, any instance of a programmer-defined class is a container object.

28. Give one example of an object that is NOT a **container** object:

Here are several examples: an **integer**, a **float**, **None**, **True**, **False**.

29. True or false: When an object is mutated, it no longer refers to the same object to which it referred prior to the mutating. True False (circle your choice)

30. Consider the following statements:

```
c1 = rg.Circle(rg.Point(200, 200), 25)
c2 = c1
```

At this point, how many **rg.Circle** objects have been constructed? 1 2 (circle your choice)

31. Continuing the previous problem, consider an additional statement that follows the preceding two statements:

```
c1.radius = 77
```

After the above statement executes, the variable **c1** refers to the same object to which it referred prior to this statement. True False (circle your choice)

32. Continuing the previous problems:

- What is the value of **c1**'s radius after the statement in the previous problem executes?
- What is the value of **c2**'s radius after the statement in the previous problem executes?

25 77 (circle your choice)

25 77 (circle your choice)

33. Which of the following two statements mutates an object? (Circle your choice.)

`numbers1 = numbers2`

`numbers1[0] = numbers2[0]`

34. Mutable objects are good because:

They allow for efficient use of space and hence time – passing a mutable object to a function allows the function to change the “insides” of the object without having to take the space and time to make a copy of the object. As such, it is an efficient way to send information back to the caller.

35. Explain briefly why mutable objects are dangerous.

When the caller sends an object to a function, the caller may not expect the function to modify the object in any way. If the function does an unexpected mutation, that may cause the caller to fail even if the function returns the correct value. If the object is immutable, no such danger exists – the caller can be certain that the object is unchanged when the function returns control to the caller. (This is not 100% accurate, since immutable objects like tuples can contain mutable objects like lists inside them, but the spirit of this answer is correct.)

36. What is the difference between the following two expressions?

`numbers[3]` `numbers = [3]` The expression on the left refers to the index 3 item in the sequence called *numbers*. It refers to that item but changes nothing (of itself). The statement on the right sets the variable called *numbers* to a list containing a single item (the number 3).

37. Consider the code shown to the right.

When Location 1 is reached the first time:

- What is the value of *miles*?

333

- What is the value of *self*?

The object to which *car1* in the *cars* function refers.

When Location 1 is reached the second time:

- What is the value of *miles*?

200

- What is the value of *self*?

The object to which *car2* in the *cars* function refers.

What does the code print when it runs?

10333 700

```

class Car(object):

    def __init__(self, m):
        self.mileage = m

    def drive(self, miles):
        #### Location 1
        self.mileage = self.mileage + miles

def cars():
    car1 = Car(10000)
    car2 = Car(500)

    car1.drive(333)
    car2.drive(200)
    print(car1.mileage, car2.mileage)

cars()

```

38. Consider the following four statements (which use the ***Point*** class you wrote in an exercise):

```
p1 = Point(4, 5)
p2 = Point(p1.x, p1.y)
p3 = p1
p4 = p2
```

At this point, how many ***Point*** objects have been constructed?
(circle your choice)

1 2 3 4

39. Consider the code snippet to the right. Trace the execution of the code snippet and ***draw a box-and-pointer diagram on the page to the right of this page*** as you trace the code's execution.

After the code snippet is executed, what are the values of the variables? (Write your answer in the spaces provided below.)

p1.x = 99

p1.y = 5

p2.x = 4

p2.y = 600

p3.x = 99

p3.y = 5

p4.x = 42

p4.y = 42

```
p1 = rg.Point(4, 5)
p2 = rg.Point(p1.x, p1.y)
p3 = p1
p4 = p2

p3.x = 99
p4.y = 600
p4 = rg.Point(42, 42)
```

Reminder:

You must ***draw*** (on the page to the right of this page) a
***** box-and-pointer *****
diagram for this problem.

Draw your box-and-pointer diagram on this page.

Color is used in the following to match the color-coded copy of the code shown to the right. That is:

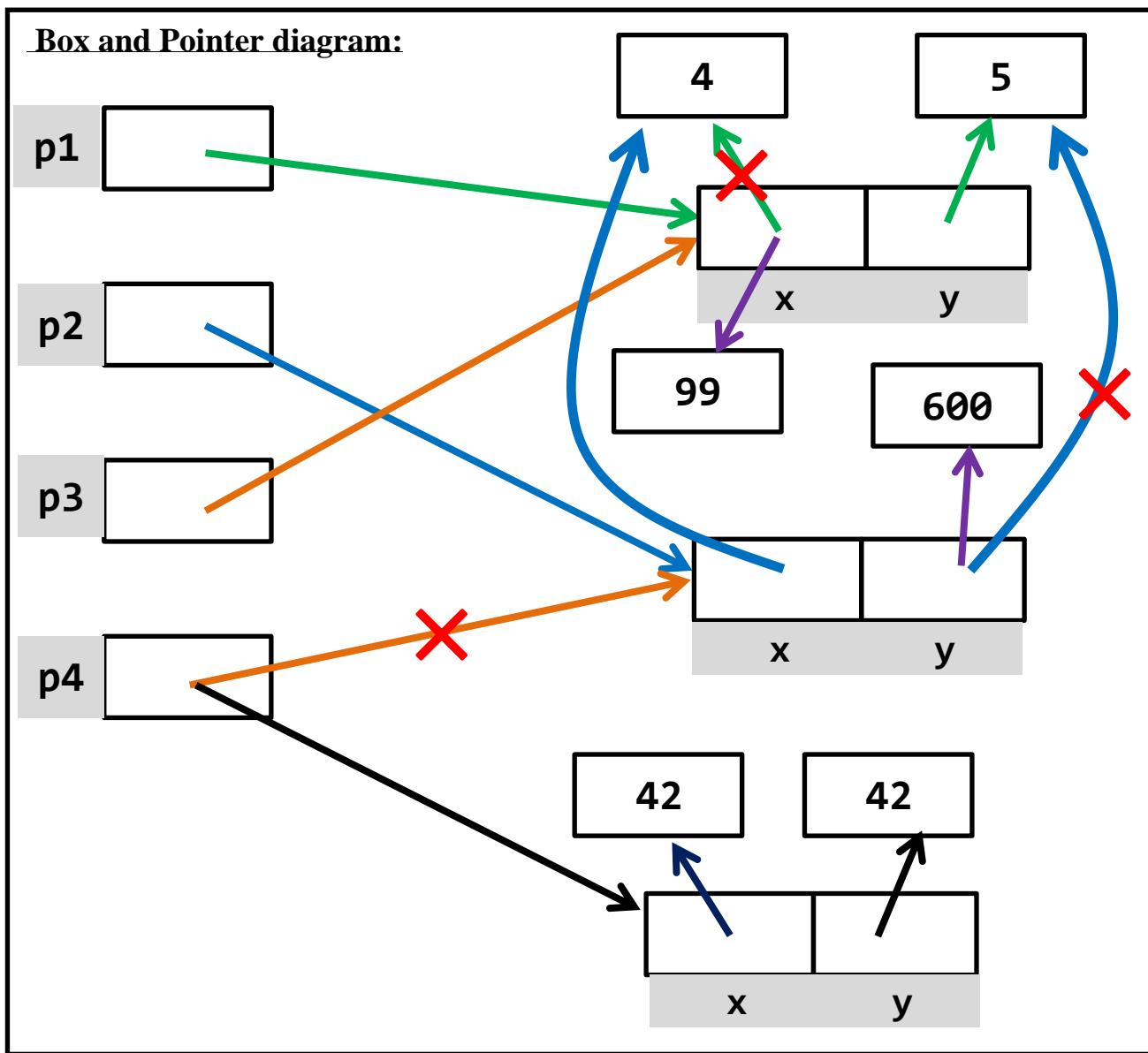
- The arrows for the 1st line of code are in green.
- The arrows for the 2nd line of code are in blue.
- The arrows for the 3rd and 4th lines of code are in orange.
- The arrows for the 5th and 6th lines of code are in purple.
- The arrows for the 7th line of code are in black.

```

p1 = rg.Point(4, 5)
p2 = rg.Point(p1.x, p1.y)
p3 = p1
p4 = p2

p3.x = 99
p4.y = 600
p4 = rg.Point(42, 42)

```



40. Assume that there is a class named **Elevator** whose constructor:

- requires two arguments: the number of floors in the Elevator's building, and the current floor at which the Elevator resides.
- and stores those arguments in instance variables named **num_floors** and **current_floor**, respectively.

Assume further that **Elevator** objects have a **go_to_floor** method that takes a positive argument **N** that is the floor to which the Elevator should move, and moves the Elevator to floor **N** unless **N** is greater than the Elevator's number of floors (in which case the Elevator remains at its current floor).

- a. Write code that would construct an **Elevator** object for a building with **12** floors, with the Elevator starting at floor **5**.

```
elevator = Elevator(12, 5)
```

(you can use any name you want where I have chosen **elevator**)

- b. Write code that would use the **go_to_floor** method to move the Elevator from part (a) to the floor that is **3** floors higher than its current floor.

```
elevator.go_to_floor(elevator.current_floor + 3)
```

41. Continuing the previous problem, implement the methods in the **Elevator** class by filling in the blanks in the following:

```
class Elevator(object):
```

```
    def __init__(self, number_of_floors, current_floor):
        # Write code here that implements the __init__ method.
        self.num_floors = number_of_floors
        self.current_floor = current_floor
```

You can use any names you wish for the arguments after **self**, as long as you use them again appropriately in the succeeding lines. The instance variables (**self.num_floors** and **self.current_floor**) must be named exactly as written.

```
def go_to_floor(self, floor):
    # Write code here that implements the go_to_floor method.
    if floor <= self.num_floors:
        self.current_floor = floor
```

You can use any name you wish for the **floor** argument after **self**, as long as you use it again appropriately in the succeeding lines. The code assumes that floors are numbered starting at 1 (not at 0), as suggested by the problem statement.

42. Consider a function named ***blah*** that takes a *list of numbers* as its sole argument. For each of the following possible specifications for what ***blah*** returns:

Circle **Yes** if the code for ***blah*** would require a loop.

Circle **No** if the code for ***blah*** would NOT require a loop.

If ***blah*** returns:

- a. The smallest number in the list. Yes No
- b. The second smallest number in the list. Yes No
- c. The second number in the list. Yes No
- d. The first positive number in the list, or -1 if there is no positive number in the list. Yes No
- e. ***True*** if the first number in the list is positive, else ***False***. Yes No
- f. ***True*** if the list contains no positive numbers, else ***False***. Yes No
- g. The average of the positive numbers in the list. Yes No
- h. The number of numbers in the list. Yes No
- i. The number of positive numbers in the list. Yes No
- j. The number in the middle of the list. Yes No