Test 2 – Practice Problems for the Paper-and-Pencil portion
Solution

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)
```

Prints:

```
main   1   5   3
foo   1   5    3
foo   2   66   77   88   99
main   2   5   3
```

```
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)
```

Prints:

```
main   1   5   3
foo 1   5    3
foo 2   66   77   88   99
main   2   5   3
```

```
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   3    5
foo 2   66   77   88   99
main   2   5   3
```

Note: extra spaces have been inserted in the above to make the output more readable.

Also ***note*** how part of the third problem has a ***DIFFERENT answer*** than the first two problems.

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

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

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

- 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*.)

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 of the code.

```python
def main():
    a = alpha()

    print()
    b = beta()

    print()
    g = gamma()

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


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


def beta():
    print("Beta!")
    return 15 + alpha()


def gamma():
    print("Gamma!", alpha(), beta())
    return alpha() + beta() + alpha()


main()
```

Output:

```
Alpha!

Beta!
Alpha!

Alpha!
Beta!
Alpha!
Gamma!   7   22
Alpha!
Beta!
Alpha!
Alpha!

main!  7   22   36
```

4. 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:

```
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
```

5. Consider a function whose name is ***last_n_reversed*** that takes two arguments: a string ***s*** and a nonnegative integer ***n***. It returns a string that is the last ***n*** characters of the string ***s***, in reverse order of how they appear in ***s***.

   Here is a code snipped that illustrates a sample run of the function:

   ```
   my_string = 'Ada Lovelace'
   answer = last_n_reversed(my_string, 6)
   print(answer)
   ```

   would print **ecalev** in the Console.

   Write a complete implementation, including the header (def) line, of the above ***last_n_reversed*** function.

   ```
   def last_n_reversed(s, n):
       result = ''
       last = len(s) - 1
       for k in range(n):
           result = result + s[last - k]
       return result
   ```

6. Consider a function whose name is ***reverse_n*** that takes two arguments: a list ***s*** and a nonnegative integer ***n*** that is less than half of the length of the list. It mutates the list ***s*** by swapping the first ***n*** items in the list with the last ***n*** items in the list.

   Here is a code snipped that illustrates a sample run of the function:

   ```
   my_list = [10, 64, 33, 20, 82, 90, 44, 50, 37, 100, 4]
   answer = reverse_n(my_list, 3)
   print(my_list)
   ```

   would print `[4, 100, 37, 20, 82, 90, 44, 50, 33, 64, 10]` in the Console.

   Write a complete implementation, including the header (def) line, of the above ***reverse_n*** function.

   ```
   def reverse_n(s, n):
       last = len(s) - 1
       for k in range(n):
           temp = s[k]
           s[k] = s[last - k]
           s[last - k] = temp
   ```

7. Consider the following two candidate function definitions:

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

[✓
```python
def foo(x):
    print(x)
```
]

   a. Which is "better"? Circle the better function.

   b. Briefly explain why you circled the one you did.

   The second form allows the caller of the function to print ANYTHING, while the first is useful only for printing `'hello'`.

8. True or false: ***Variables are REFERENCES to objects.*** [✓ **True**] **False** (circle your choice)

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

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

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

    Here are several examples: a *list*, a *tuple*, a rg.Circle, a Point, an rg.*window*

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

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

13. 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)

14. Consider the following statements:

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

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

15. 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)

16. Continuing the previous problems:

- What is the value of ***c1***'s radius after the statement in the previous problem executes? **25** [✓ **77**] (circle your choice)
- What is the value of ***c2***'s radius after the statement in the previous problem executes? **25** [✓ **77**] (circle your choice)

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

```
numbers1 = numbers2
```

[✓ `numbers1[0] = numbers2[0]`]

18. 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.**

19. 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. 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.**

20. 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).

21. In Session 9, you implemented a ***Point*** class. Recall that a Point object has instance variables **x** and **y** for its x and y coordinates

Consider the code snippets 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.)

Suggestion: Draw a box-and-pointer diagram to solve this problem, even though the problem does not require you to do so.

```
def main():
    p1 = Point(11, 12)
    p2 = Point(77, 88)
    p3 = foo(p1, p2)
    print(p1.x, p1.y)
    print(p2.x, p2.y)
    print(p3.x, p3.y)

def foo(p1, p2):
    p1 = Point(0, 0)
    p1.x = 100
    p2.y = 200
    p3 = Point(p2.x, p1.y)
    return p3
```

```
def main():
    a = [1, 2, 3]
    b = [100, 200, 300]
    c = foofoo(a, b)
    print(a)
    print(b)
    print(c)

def foofoo(a, b):
    a = [11, 22, 33]
    a[0] = 777
    b[0] = 888
    x = [a[1], b[1]]
    return x
```

$\swarrow$

Prints:

```
11   12

77  200

77    0
```

$\swarrow$

Prints:

```
[1, 2, 3]

[888, 200, 300]

[22, 200]
```

22. In Session 9, you implemented a ***Point*** class. Recall that a Point object has instance variables **x** and **y** for its x and y coordinates.

Here, you will implement a portion of a class called ***TwoPoints***, described as follows:

- The ***TwoPoints*** constructor takes 2 arguments, each a ***Point*** object.
- The ***TwoPoints*** class has a method called **swap()**. It swaps the two points that a ***TwoPoints*** object has.
- The ***TwoPoints*** class has a method called **number_of_swaps()** that returns the number of times the TwoPoints object has called its **swaps()** method.

In this column, write code that would TEST the TwoPoints class.

```
p1 = Point(10, 20)
p2 = Point(88, 44)
tp = TwoPoints(p1, p2)
print('Expected:', p1, p2)
print('Actual:   ', tp.p1, tp.p2)

tp.swap()
print('Expected:', p2, p1)
print('Actual:   ', tp.p1, tp.p2)

tp.swap()
print('Expected:', p1, p2)
print('Actual:   ', tp.p1, tp.p2)

print('Expected:', 2)
print('Actual:   ',
      tp.number_of_swaps())
```

Many other answers are possible as well. All correct answers will construct at least one ***TwoPoints*** object, reference its instance variables, call ***swap*** and then reference the instance variables again, and call ***number_of_swaps***, checking that the returned value is correct.

In this column, write the IMPLEMENTATION of the TwoPoints class.

```
class TwoPoints(object):

    def __init__(self, p1, p2):
        self.p1 = Point(p1.x, p1.y)
        self.p2 = Point(p2.x, p2.y)
        self.nswaps = 0

    def swap(self):
        temp = self.p1
        self.p1 = self.p2
        self.p2 = temp
        self.nswaps = self.nswaps + 1

    def number_of_swaps(self):
        return self.nswaps
```

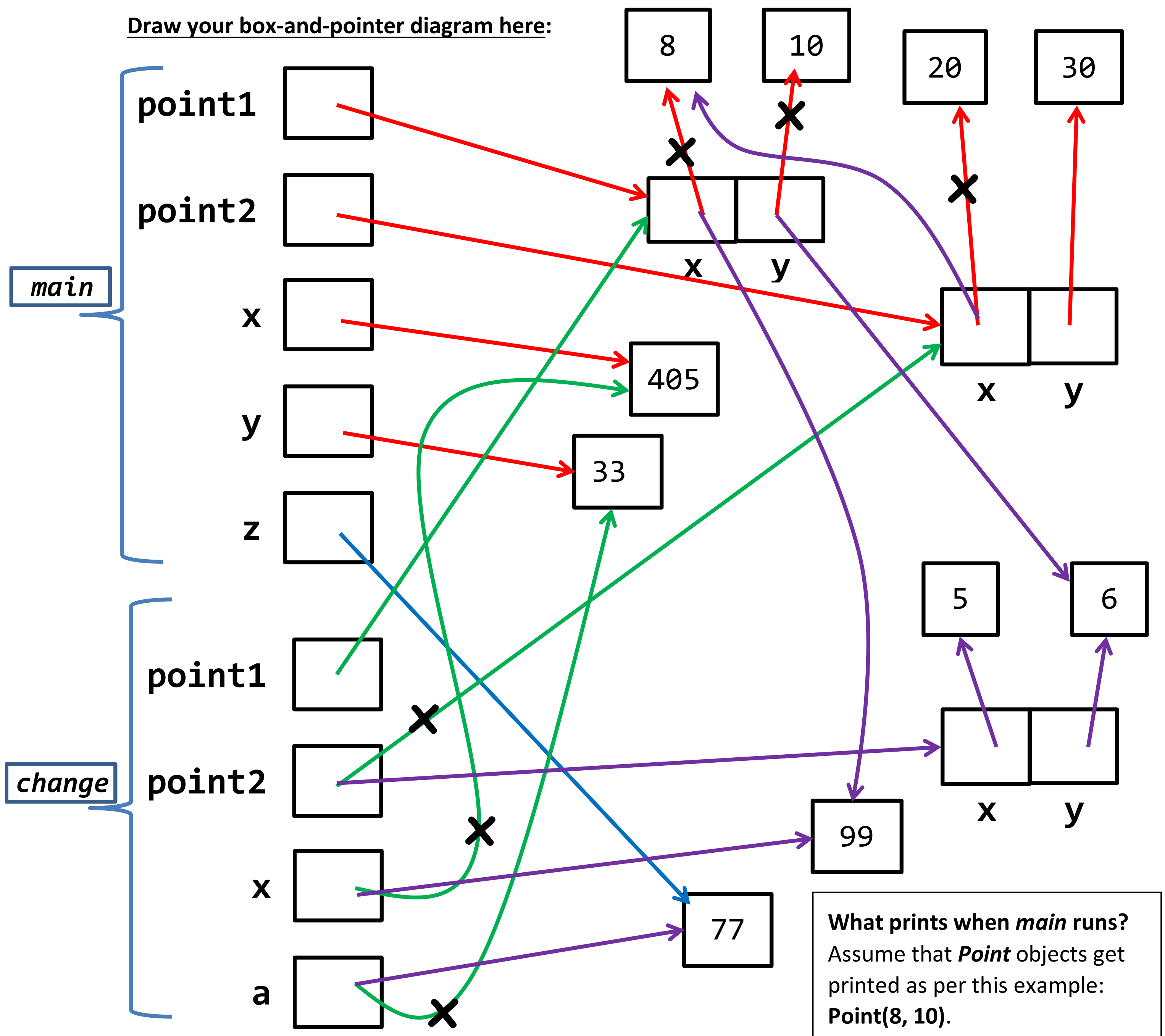
There are other correct answers possible. All correct answers will set instance variables to store the two ***Point*** objects that are arguments for ***__init__***, though it is equally correct to clone the arguments or not (the specification is ambiguous on that issue). All correct solutions will have an instance variable for the number of swaps, initializing it in ***__init__***, setting it in ***swap***, and returning it in ***number_of_swaps***.

23. In Session 9, you implemented a ***Point*** class. Recall that a ***Point*** object has instance variables ***x*** and ***y*** for its x and y coordinates.

Consider the code in the box below. On the **next** page, draw the ***box-and-pointer diagram*** for what happens when ***main*** runs. Also on the next page, show what the code would ***print*** when ***main*** runs.

```
def main():
    point1 = Point(8, 10)
    point2 = Point(20, 30)
    x = 405
    y = 33

    print('Before:', point1, point2, x, y)

    z = change(point1, point2, x, y)

    print('After:', point1, point2, x, y, z)


def change(point1, point2, x, a):
    print('Within 1:', point1, point2, x, a)
    point2.x = point1.x
    point2 = Point(5, 6)
    point1.y = point2.y
    x = 99
    point1.x = x
    a = 77

    print('Within 2:', point1, point2, x, a)

    return a
```

Draw your box-and-pointer diagram here:



What prints when *main* runs? Assume that ***Point*** objects get printed as per this example: **Point(8, 10)**.

Before: The ***RED*** lines reflect the execution of the lines in ***main*** before the call to function ***change***. Therefore, what gets printed BEFORE the call to ***change*** is:

```
Point(8, 10)    Point(20, 30)    405    33
```

Within: The ***GREEN*** lines reflect the execution of the call to function ***change***. Thus what gets printed at ***Within 1:*** is `Point(8, 10)    Point(20, 30)    405    33`

WITHIN the call to ***change*** (at the end of that function, i.e., when ***Within 2:*** is printed) is:

```
Point(99, 6)    Point(5, 6)    99    77
```

After: The ***BLUE*** line reflects the execution of the return from ***change*** and the assignment to ***z*** in function ***main***. Therefore, what gets printed AFTER the call to ***change*** is:

```
Point(99, 6)    Point(8, 30)    405    33    77
```

From the picture on the previous page, we see that:

What prints when *main* runs?
Assume that ***Point*** objects get printed as per this example: **Point(8, 10)**.

Before: `Point(8, 10)    Point(20, 30)    405    33`

Within 1: `Point(8, 10)    Point(20, 30)    405    33`

Within 2: `Point(99, 6)    Point(5, 6)    99    77`

After: `Point(99, 6)    Point(8, 30)    405    33    77`