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

SOLUTION

Test 2 will assess material covered in Sessions 1 through 14 (but NOT material from Session 15). It will draw problems especially from the following concepts, where the numbers in the brackets at the beginning of the item are problems that let you practice that concept:

- `[1, 2, 3, 7, 22]` ***Scope***, including scope inside a method.
- `[2, 4]` Function calls and returns – ***flow of control***. Including calls within expressions (e.g. `print(foo1(…), foo2(…))` or `x = foo1(..) + foo2(…)`.
- `[4, 6, 18]` ***Range expressions***: All 3 forms.
- `[5, 7 - 13, 18]` ***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 `x[...] = ....`
- `[13 - 17]` ***Concatenating*** items to a sequence.
- `[18 - 21]` Write simple functions that ***loop through a sequence*** and ***access*** (e.g. sum/count), ***find***, or ***accumulate***.
- `[23 - 36]` ***References*** and related concepts.
- `[38, 40]` ***Box and pointer diagrams***.
- `[17, 26 - 38, 40]` ***Mutation*** of a list / object by a function.
- `[10 - 12, 33, 34]` Fact that tuples and strings are ***immutable***. What that means.
- `[39]` ***Constructing*** objects. Using ***instance variables***. Calling ***methods***. Doing all these inside a class as well as outside of the class.
- `[37]` What ***SELF*** is. How to use it.

The actual test'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 Test 2.

Pay special attention to Problems 2 and 4, since they summarize many of the 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)
```

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
```

2. Consider what would happen if the code below were to run. It is a contrived example with poor style but will run without errors.

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

def foo(a):
    #### Location 3
    a = 6
    m = 31
    #### Location 4
    m1 = Mini(10)
    m2 = Mini(22)
    m1.m = m1.m + m2.m
    # Location 5

def main():
    a = 44
    #### Location 6
    foo(a)
    #### Location 7

main()
```

In the table to the right, indicate the values of the specified variables at the specified locations in the code, as those locations are encountered during the run. If a variable is undefined at that location, put an ***X*** in its box to indicate so.

For example, the run starts in ***main***, as usual. So, the first of the seven locations to be encountered is Location 6. At Location 6, the variable ***a*** has value ***44***, and all other variables are undefined. Hence, **Location 6** would be filled out as we have done in the table to the left.

Location	Variable	is	Value	Variable	is	Value
When **Location 1** is encountered the 1st time, the value of:	a	is	10	self.m	is	X
	m	is	X	m1.m	is	X
	self.a	is	X	m2.m	is	X
When **Location 1** is encountered the 2nd time, the value of:	a	is	22	self.m	is	X
	m	is	X	m1.m	is	X
	self.a	is	X	m2.m	is	X
When **Location 2** is encountered the 1st time, the value of:	a	is	10	self.m	is	15
	m	is	X	m1.m	is	X
	self.a	is	3	m2.m	is	X
When **Location 2** is encountered the 2nd time, the value of:	a	is	22	self.m	is	27
	m	is	X	m1.m	is	X
	self.a	is	3	m2.m	is	X
When **Location 3** is encountered, the value of:	a	is	44	self.m	is	X
	m	is	X	m1.m	is	X
	self.a	is	X	m2.m	is	X
When **Location 4** is encountered, the value of:	a	is	6	self.m	is	X
	m	is	31	m1.m	is	X
	self.a	is	X	m2.m	is	X
When **Location 5** is encountered, the value of:	a	is	6	self.m	is	X
	m	is	31	m1.m	is	42
	self.a	is	X	m2.m	is	27
When **Location 6** is encountered, the value of:	a	is	44	self.m	is	X
	m	is	X	m1.m	is	X
	self.a	is	X	m2.m	is	X
When **Location 7** is encountered, the value of:	a	is	44	self.m	is	X
	m	is	X	m1.m	is	X
	self.a	is	X	m2.m	is	X

3. 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*.)

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 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
```

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 not 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(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(T[0])` would print: **4**
   - `print(T[2])` 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(s[0])` would print: **h**
   - `print(s[4])` would print: **o**
   - `print(s[len(s)])` would print: **ERROR**

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 the list `X = []` and the statement (in this order)s:

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

Would the above statements would generate an exception (error)? [✓ Yes] or No (circle your answer)

14. 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,)
```

15. 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'
```

16. 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']
```

17. 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')
```

18. 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])
```

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

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

Alternative:

```
for item in X:
    print(item)
```

- 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])
```

There are other alternatives as well, 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 this 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])
```

19. 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
```

20. 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('abcdefgh')         returns   ['a', 'c', 'e', 'g']
```

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

The statement in the above:

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

is better written as:

```
items.append(sequence[k])
```

21. 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
```

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

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

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

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

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

26. 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*

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

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

28. 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)

29. 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)

30. 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)

31. 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)

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. 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.

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.

```
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()
```

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** 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** refers.

What does the code print when it runs?

```
10333   700
```

38. 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.)

```
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
```

Prints:

```
11   12

77  200

77    0
```

Prints:

```
[1, 2, 3]

[888, 200, 300]

[22, 200]
```

39. 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, and stores them.
- The ***TwoPoints*** class has a method called ***clone()***. It returns a new ***TwoPoints*** object whose two ***Point*** objects are clones of the ***Point*** objects that the ***TwoPoints*** object has.
- The ***TwoPoints*** class has a method called ***swap()***. It swaps the two points that the ***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 ***swap()*** method.

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

```
p1 = Point(10, 20)
p2 = Point(88, 44)
tp = TwoPoints(p1, p2)

# Testing construction (__init__):
print('Expected:', p1, p2)
print('Actual:   ', tp.p1, tp.p2)

# Testing clone:
tp2 = tp.clone()    # rest of this test left to you

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

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

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

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

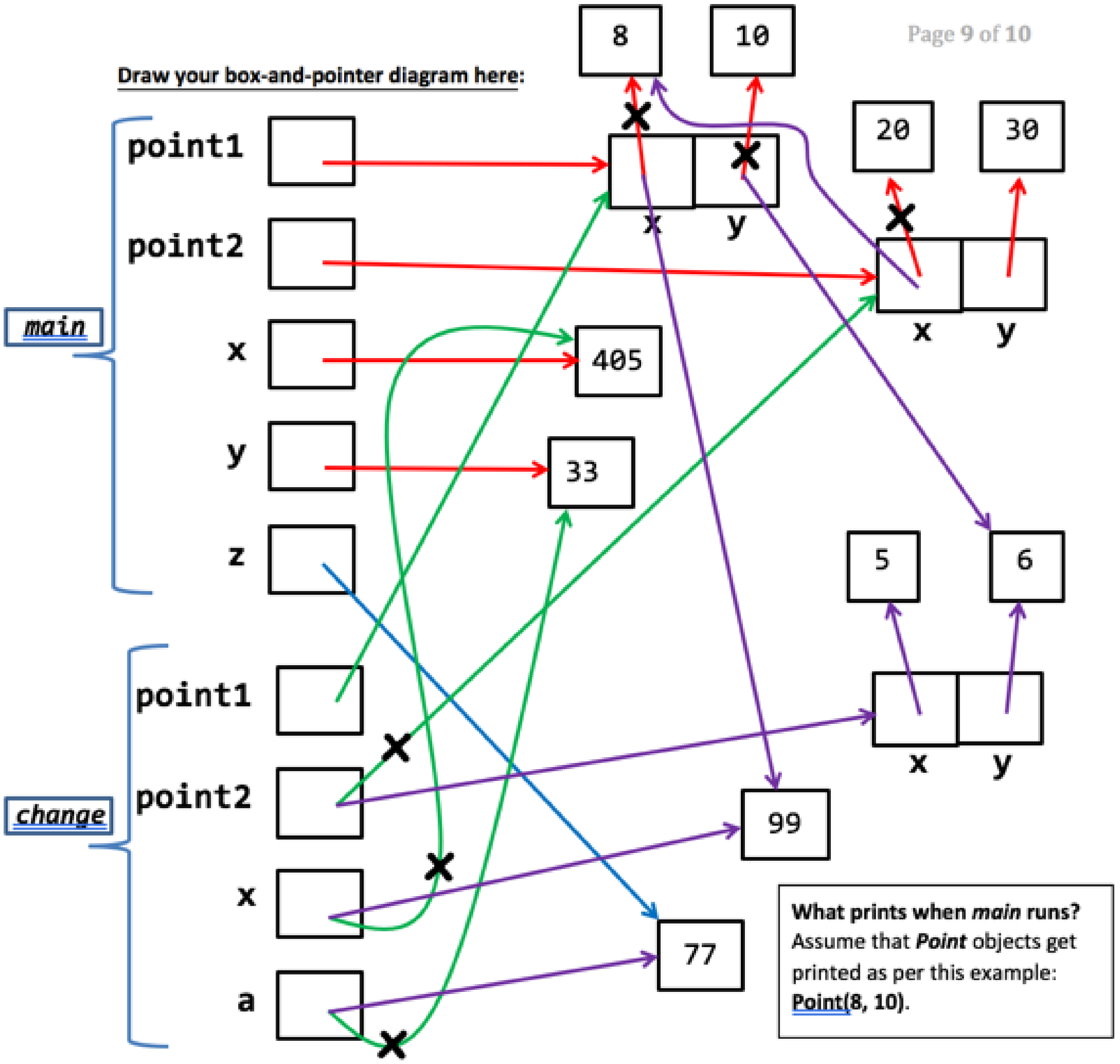
```
class TwoPoints(object):

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

    def clone(self):
        p1 = Point(self.p1.x,
                   self.p1.y)
        p2 = Point(self.p2.x,
                   self.p2.y)
        return TwoPoints(p1, p2)

    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
```

40. 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`

The ***PURPLE*** lines reflect the execution of the lines in ***change***. Therefore, what gets printed 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`