CS 1110 Final Exam Solutions, May 2024

1. [10 points] **Strings.** Implement the following function. Do not use iteration in your solution.

```
def outside_markers(text, marker):
    """
    Returns: a string found outside the first and last instance of `marker` in `text`
             If `marker` is not found, return the entire string.
             If `marker` is found only once, return the marker.

    Preconditions
       text   [str]: a string with 0 or more instances of `marker`
       marker [str]: a string of length 1

    Examples:
      outside_markers("ab+c+de+f++g", "+")   returns "abg"
      outside_markers("blah park", "a")      returns "blrk"
      outside_markers("xabcdefgx", "x")    returns ""
      outside_markers("hi my name is", "z") returns "hi my name is"
      outside_markers("abcdef+ghijkl", "+")  returns "+"
    """

    n = text.count(marker)
    if n == 0:
        return text
    elif n == 1:
        return marker
    first_i = text.find(marker)
    last_i = text.rfind(marker)

    outside = text[0:first_i] + text[last_i+1:]
    return outside


    # Alternate Solution #1:
    loc1 = text.find(marker)
    loc2 = text.rfind(marker)
    if loc1 == -1 and loc2 == -1:
        return text
    if loc1 == loc2:
        return marker
    return text[:loc1]+text[(loc2+1):]
```

```
# Alternate Solution #2:
parts = text.split(marker)
if len(parts) == 1:   # not found
    return text
elif len(parts) == 2: # found once
    return marker
else:
    return parts[0] + parts[-1]
```

2. [10 points] **Blue Screen of Death.** An image can be represented as a 2-dimensional (aka "nested") list of pixels. Each pixel can be represented as a list with three elements representing a red, green, and blue value from 0 to 255. For example `[0, 0, 0]` is black, `[255, 0, 0]` is red, and `[255, 255, 255]` is white. Implement the following function according to its specification. You may only use Python taught in CS 1110. (No list comprehension allowed.)

```
def make_colored_image(width, height, r, g, b):
    """ Returns: a 2d list representing an image of size `width` by `height`.
    Each element in the image is a unique list with three elements, representing
    a pixel with RGB values specified by parameters r, g, and b. The elements
    representing GRB pixels in the 2d list should have unique identifiers (i.e.,
    there should be multiple, distinct lists with three elements, not just one).

    Examples:
    make_colored_image(2,2,255,0,0) returns
       [[[255, 0, 0], [255, 0, 0]], [[255, 0, 0], [255, 0, 0]]]

    make_colored_image(1,3,128,128,0) returns
       [[[128, 128, 0]], [[128, 128, 0]], [[128, 128, 0]]]

    Preconditions:
       width, height: int > 0
       r, g, b: int >= 0 and <= 255                             """

    img = []
    for i in range(height):
        row = []
        for j in range(width):
            row.append([r,g,b])
        img.append(row)
    return img
```

3. [10 points] **Dictionaries.** Implement this function according to its specification:

```
def add_unique(d, k, elem):
    """Given a dictionary `d` with values that are lists of elements,
    this function adds `elem` to the list associated with key `k`.
    If `elem` is already present in the list associated with key `k`,
    this function does nothing.
    Returns: Nothing

              -- EXAMPLES --
    parameters             after the function ends, d will be:
    -------------------------------------------------------------
    d: {}
    k: 'P'                 -->   d: {'P': ['hi']}
    elem: 'hi'

    d: {'P': ['hi']}
    k: 'P'                 -->   d: {'P': ['hi']}
    elem: 'hi'

    d: {'P': ['hi']}
    k: 'P'                 -->   d: {'P': ['hi','bye']}
    elem: 'bye'

    d: {'P': ['hi','bye']}
    k: 'U'                 -->   d: {'P': ['hi','bye'],
    elem: 'bye'                     'U': ['bye'] }

    Preconditions:
      d is a dictionary
      k is an acceptable dictionary key                       """


    if k in d:
        elems = d[k]
        if elem not in elems:
            elems.append(elem)
    else:
        d[k] = [elem]
```

4. **CurMu** is a made-up language used to tell a computer on how to style text. Correct **CurMu** *always* has the pattern: `Xcccc@` and follows these rules:

   1. The letter `X` at the beginning must be an uppercase letter. It represents a tag, like `P` for plain, `B` for bold, `I` for italicized, or `U` for underline.
   2. Following the tag, there is a non-empty sequence of characters (`cccc`) that should be styled according to the tag. This sequence must not contain any capital letters or the `@` symbol.
   3. The styling designation for a sequence of characters ends with the `@` symbol.
   4. You can repeat this pattern multiple times in a text to apply different styles to different sequences of characters. Every part of the text must be styled in some way, even if is just plain. Each character or sequence of characters may only have 1 style applied to it.

   Here are some examples of **CurrMu** and what the corresponding text should look like:

CurMu text	What it the text should look like:
`Pplain@`	plain
`Bbold!@`	**bold!**
`Iitalicize?@`	*italicize?*
`Pplain. @Bbold.. @Iitalicize. @Uunderline. @`	plain. **bold..** *italicize.* underline.
`Pthis text is plain.@Bthis text is bold.@`	this text is plain.**this text is bold.**
`Bno! @Pi am @Inot@P looking at @Uyou.@`	**no!** i am *not* looking at you.

   (a) [1 point] To underline the word happy (*i.e.*, happy), how do you write that in **CurMu**?

   Correct Answer: `Uhappy@`

   (b) [3 points] Which of the following are correct **CurMu**?

   (A) `UB_underline_and_bold@@`

   (B) `Uitalicize?@`

   (C) `plain text please@`

   (D) `Uit's Mundy, baby!@`

   (E) `I.?!@`

   (F) `IB@`

Circle all that apply: A B C D E F None

Correct Answer: B & E

(c) [14 points] Implement the function below according to its specification. To receive full credit, **you must use the function** `add_unique` from the previous question. (Assume you have imported a correct implementation of the function.)

```
def get_styles(text):
    """
    Returns a dictionary that associates strings to their styles.

    The key is a single letter tag (like 'P', 'B', 'U', 'I', etc.).
    The value is a list of unique strings that should be styled by that tag.

    Examples:

    get_styles("Bno! @P i am @Inot @P looking at @Uyou.@")
       returns {'B': ['no! '],
                'P': [' i am ', ' looking at '],
                'I': ['not '],
                'U': ['you.']}

    get_styles("Bno! i @Pam @Bnot!@Bnot!@")
       returns {'B': ['no! i ', 'not!'],
                'P': ['am '] }

    Preconditions: `text` is correctly-formatted CurMu
    """
```

```
    d = {}
    key = text[0]
    chars = ""
    for i in range(1,len(text)):
        c = text[i]
        if c == '@':
            add_elem_to_d(d, key, chars)
        elif c.isupper():
            key = c
            chars = ""
        else:
            chars += c
    return d
```

```
# Alternate Solution #1
key = ""
value = ""
d = {}
for char in text:
    if char.isupper():
        key = char
    elif char == "@":
        add_unique(d,key, value)
        key = ""
        value = ""
    else:
        value += char
return d

# Alternate Solution #2
d = {}
curr_index = 0
while curr_index < len(text):
    at_loc = text.find('@',curr_index)
    word = text[curr_index+1:at_loc]
    add_unique(d, text[curr_index], word)
    curr_index = at_loc+1
return d

# Alternate Solution #3
d = {}
tokens = text.split('@')[:-1]
for token in tokens:
    style = token[0]
    seq = token[1:]
    add_unique(d, style, seq)
return d

# Alternate Solution #4
d = {}
for i in range(len(text)):
    if text[i].isupper():
        index = text.find("@",i)
        add_unique(d, text[i],text[i+1:index]
return d
```

5. [14 points] **Recursion.** Consider a world in which there are only 2 eye colors and eye color is a trait that skips every generation. A `Person` has a different color eyes from their parents, but the same color eyes as their grandparents. Let `Person` be a class as defined below:

```
class Person:
    """An instance represents a human with an eye color and up to 2 parents.

    Initially, eye_color is "unknown". After a family is created, eye_color is
    assigned for the family by calling set_eye_color() from the root of the tree.

    Instance attributes:
       name             [str] - unique non-empty name of a person
       eye_color        [str] - the color of the person's eyes or "unknown"
       p1    [Person or None] - parent 1
       p2    [Person or None] - parent 2
    """
```

Implement the method `set_eye_color`, which sets the eye colors for a family, as illustrated below:

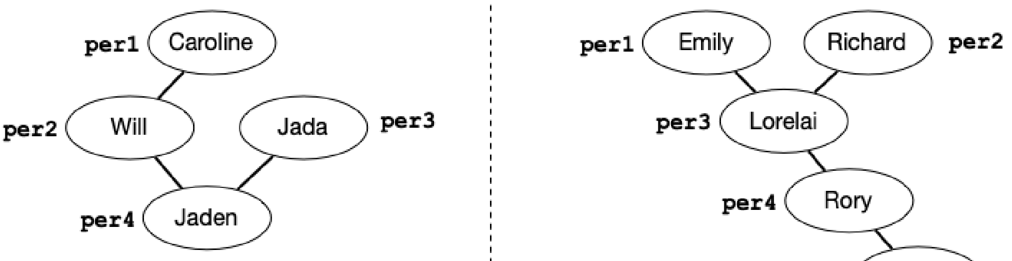
EXAMPLE 1 | **EXAMPLE 2**

Suppose you create the following Person objects:

```
per1 = Person("Caroline")
per2 = Person("Will", None, per1)
per3 = Person("Jada")
per4 = Person("Jaden", per2, per3)
```

```
per1 = Person("Emily")
per2 = Person("Richard")
per3 = Person("Lorelai", per1, per2)
per4 = Person("Rory", per3)
per5 = Person("Baby", per4)
```

You can picture their initial family tree to look like this, conceptually:

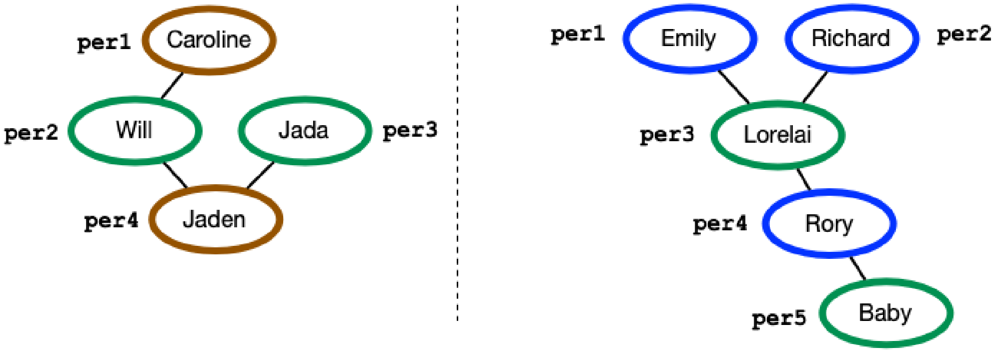


Next you execute the following method to set all the eye colors in the tree:

```
per4.set_eye_color("brown", "green")
```

```
per5.set_eye_color("green", "blue")
```

You can picture the eye colors of the family to be set as follows:



(Continued from previous page.)

Implement the `Person` class' instance method `set_eye_color`, making effective use of recursion.

```
def set_eye_color(self, my_color, parent_color):
    """Sets the eye_color for the entire family reachable by self.

    eye_color of self is set to my_color
    eye_color of self's parents is set to parent_color
    eye_color of self's grandparents is set to my_color
    ... and so forth (color alternates with each generation)

    Precondition:
        - everyone reachable by self currently has eye_color "unknown"
        - everyone in the family is reachable only once
        - my_color, parent_color are non-empty, unique strings
    Returns nothing.
    """

    self.eye_color = my_color
    if self.p1 != None:
        self.p1.set_eye_color(parent_color, my_color)
    if self.p2 != None:
        self.p2.set_eye_color(parent_color, my_color)
```

6. **Debugging.** Consider the following code:

```
1  class Player:
2  
3      def __init__(nam):
4          self.name = nam
5  
6  class Team:
7  
8      def __init__(team, name, player_names=None):
9          team.name = name
10         team.players = []
11         if player_names == None:
12             player_names = []
13         for pname in player_names:
14             team.addPlayer(pname)
15 
16     def makePlayer(self, name):
17         return Player(name)
18 
19     def addPlayer(self, name):
20         p = self.makePlayer(name)
21         self.players.append(p)
22         return p
23 
24 t1 = Team("Princeton Revolution", ["Angela Guan"])
```

When the above code is run in Python, the following error is reported:

```
Traceback (most recent call last):
  File "team.py", line 24, in <module>
    t1 = Team("Princeton Revolution", ["Angela Guan"])
  File "team.py", line 14, in __init__
    team.addPlayer(pname)
  File "team.py", line 20, in addPlayer
    p = self.makePlayer(name)
  File "team.py", line 17, in makePlayer
    return Player(name)
TypeError: __init__() takes 1 positional argument but 2 were given
```

(a) [3 points] Fix the code to remove only the above error. **Fix only the problem that directly causes the above error message.** Mark your fix(es) with the label **FIX1**.

need to add `self` as first parameter to `Player.__init__()` on line 2

```
26  class NumberedPlayer(Player):
27
28      def __init__(self, name, n):    # override Player method
29          super().__init__(name)
30          self.num = n
31
32  class SoccerTeam(Player):
33
34      def __init__(self, name, player_names=None):
35          super().__init__(name, player_names)
36          self.next_num = 0
37
38      def makePlayer(self, name):   # override Team method
39          p = NumberedPlayer(name, self.next_num)
40          self.next_num += 1
41          return p
42
43  t2 = SoccerTeam("Spurs", ["Timo Werner"])
```

After fixing the error, you add the above code: two subclasses and 1 new line of code that uses them. When the new code is run in Python, the following error is reported:

```
Traceback (most recent call last):
  File "team.py", line 43, in <module>
    t2 = SoccerTeam("Spurs", ["Timo Werner"])
  File "team.py", line 35, in __init__
    super().__init__(name, player_names)
TypeError: __init__() takes 2 positional arguments but 3 were given
```

(b) [3 points] Fix the code to remove only this new error. **Fix only the problem that directly causes to the new error message.** Mark your fix(es) with the label **FIX2**.

Line 32: `SoccerPlayer` should inherit from `Player` not `Team` (The argument count is off, because the wrong `__init__` method is being inherited.)

After fixing the error, you rerun the code and the following error is reported:

```
Traceback (most recent call last):
  File "team.py", line 43, in <module>
    t2 = SoccerTeam("Spurs", ["Timo Werner"])
  File "team.py", line 35, in __init__
    super().__init__(name, player_names)
  File "team.py", line 14, in __init__
    self.addPlayer(pname)
  File "team.py", line 20, in addPlayer
    p = self.makePlayer(name)
  File "team.py", line 39, in makePlayer
    p = NumberedPlayer(name, self.next_num)
AttributeError: 'SoccerTeam' object has no attribute 'next_num'
```

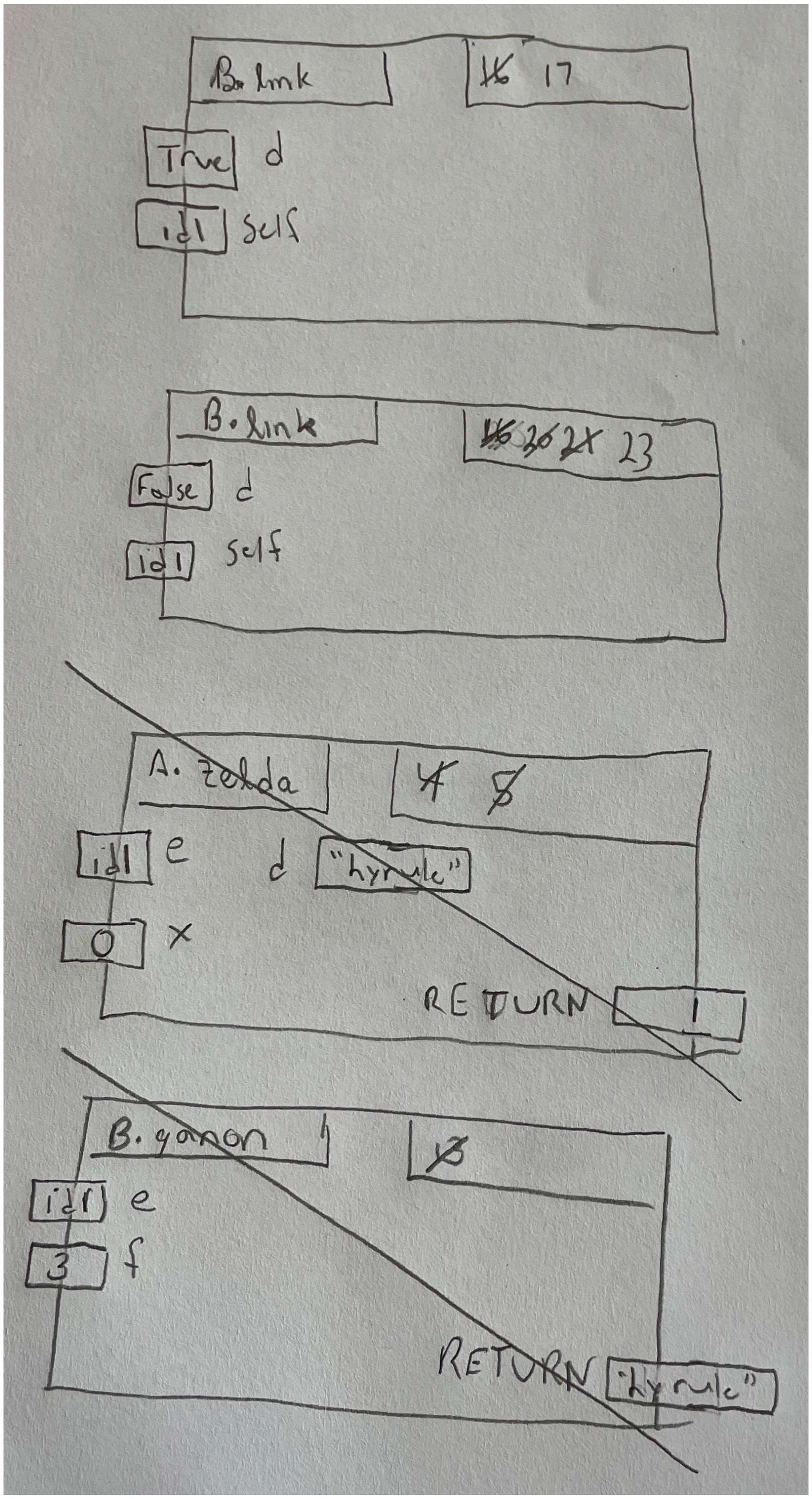
(c) [3 points] Fix the code to remove only this new error. **Fix only the problem that directly causes to the new error message.** Mark your fix(es) with the label **FIX3**.

need to swap lines 35 and 36 so that `self.next_num` exists for the init method to use.

7. [16 points] The code below runs to completion without errors. Simulate running the code, **drawing only the Call Stack**. Begin your drawings at line 27. Stop your simulation after Python finishes executing either line 17 or line 21 (whichever it reaches first) and reaches the comment `# STOP HERE` either on line 18 or 22. At the stopping point, the instruction counter of the call frame should have either the value 19 or 23. For method call frames, give the method name as `className.methodName()`, since we need to know which class's method is being called. You do not need to draw the Heap or the Global Space, but assume that `d` in the Global space is assigned the value `id1` on line 25.

```
1  class A:
2
3      def zelda(e, x):
4          d = e.ganon(3)
5          return x + 1
6
7      def ganon(self, d):
8          return 10 + d
9
10 class B(A):
11
12     def ganon(e, f):
13         return "hyrule"
14
15     def link(self, d):
16         if d:
17             self.link(False)
18             # STOP HERE
19             d = 4
20         else:
21             self.zelda(int(d))
22             # STOP HERE
23             d = 5
24
25 d = B() # d is assigned id1
26 # START DRAWING HERE
27 d.link(True)
```

CALL STACK

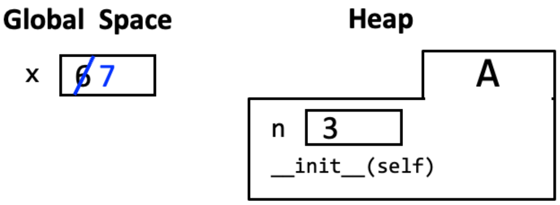

B.link
~~16~~ 17
True d
id1 self
B.link
~~16~~ ~~20~~ ~~21~~ 23
False d
id1 self
A. zelda
~~4~~ ~~5~~
id1 e
d "hyrule"
0 x
RETURN 1
B. ganon
~~3~~
id1 e
3 f
RETURN "hyrule"

8. [14 points] Implement this **function**, using a **while loop**. Do not use `break` or `continue`.

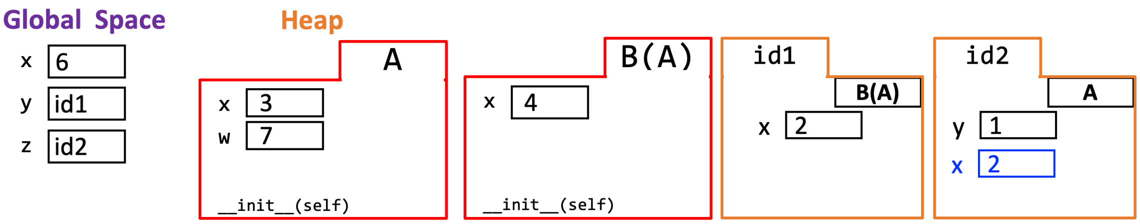
```
def process_donations(goal, donations):
    """ Given a fundraising `goal` and an int list of `donations`,
    Returns True if the goal is met and False if the goal is not met.

    ALSO, the donations list must be modified as follows:
    `donations` are processed one at a time, starting with index 0.
    If the entire donation is needed to meet the fundraising
       goal, the donation is removed from the list.
    If the goal is met using only part of the donation at index 0, the needed
       amount is subtracted from this element; the rest of the list remains untouched.

                     EXAMPLES:
    goal: 20                --> return True (goal met, 2 donations used)
    donations: [10,10,10]    donations now looks like: [10]

    goal: 10                --> return False (goal not met, all used)
    donations: [5]           donations now looks like: []

    goal: 8                 --> return True  (goal met, partial used)
    donations: [5,5,5]       donations now looks like: [2,5]

    Precondition:  goal: int >= 0
                   donations: a (possibly empty) list of positive integers """

    while goal > 0 and donations != []:
        if (donations[0] <= goal):
            goal -= donations[0]
            donations.pop(0)
        else:
            donations[0] -= goal
            goal = 0
            return True
    return goal == 0


    # Alternate Solution #1
    if len(donations) == 0:
        return goal == 0
    while len(donations) and goal > 0:
        curr_donation = donations[0]
        if goal < curr_donation:
            donations[0] -= goal
            return True
        donations.pop(0)
        if goal == curr_donation:
            return True
        goal -= curr_donation
    return False
```

9. [6 points] **Visualizing Inheritance.** For this question, you will be shown the state of memory before a single assignment statement is executed. Modify the drawing to show how memory changes after that single assignment statement has been executed. If at any point an error is thrown, please write **ERROR** next to the assignment statement; only draw the changes to memory that would occur before the error occurs.

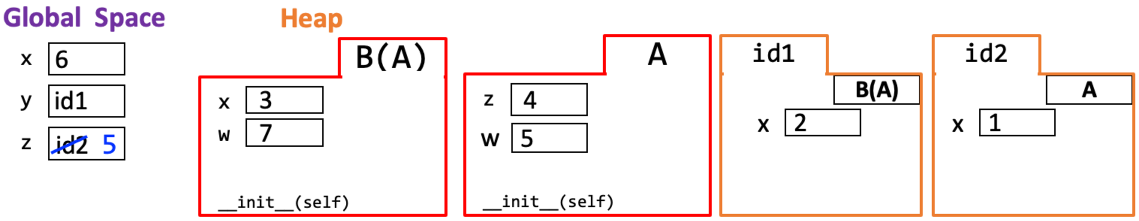
Each part is independent.
Notice: there is no Call Stack.
To the right is an example:



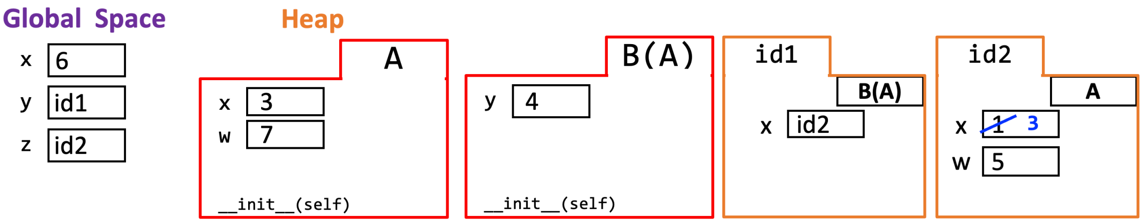
```
>>> x = x + 1
```



```
>>> z.x = y.x
```



```
>>> z = z.w
```



```
>>> y.x.x = B.x
```

10. **Multiple Choice.** Please provide **only 1 answer**. If you provide 2, we will only grade the first.

(a) [2 points] Which statement about inheritance in Python is true?

(A) It isn't possible to for a subclass to provide its own implementation of a method and also call the parent class method of the same name.

(B) The identifier of an instance object can be used to assign a value to a class attribute.

(C) A Class name can be used to access an attribute of an instance of that class.

(D) The bottom up rule looks for attributes starting with an instance folder and then moving to (possibly many) class folders.

(E) A subclass needs to implement its own `__init__` method.

Correct Answer: D

(b) [2 points] Which statement about Python is true?

(A) A Python list can only contain elements of the same data type.

(B) It's possible that Python evaluates the expression `x and y` without ever evaluating `y`.

(C) `super().super()` is one way to reach the superclass of a superclass of an object.

(D) When executing an assignment statement, Python first finds and/or creates a variable that will be assigned the value on the righthand side.

(E) Appending an element to a list changes the identifier of the list.

Correct Answer: B

(c) [2 points] Which statement about **Merge Sort** is true?

(A) The base case of the **Merge Sort** algorithm sorts two lists of size 1.

(B) In **Merge Sort**, doubling the list doubles the time it takes to sort the list.

(C) To sort 1 million integers using **Merge Sort**, Python needs room in memory to store a minimum of 2 million integers.

(D) **Merge Sort** is just as fast at sorting as Insertion Sort.

(E) **Merge Sort** is implemented with nested for loops.

Correct Answer: C

(d) [2 points] Which statement about **Linear Search** is **true**?

(A) To find the element `x` in a list of `n` elements, **Linear Search** will inspect all `n` elements.

(B) **Linear Search** can find elements in a list more quickly if the list is sorted.

(C) **Insertion Sort**'s `push_down` function is essentially **Linear Search** on a sorted list.

(D) In **Linear Search**, doubling the list size quadruples the expected time of the search.

(E) **Linear Search** is faster than **Binary Search**.

Correct Answer: C