Last Name: _____ First: _____ Netid: _____

CS 1110 Final, December 14th, 2024

This 150-minute exam has 8 questions worth a total of 100 points. Scan the whole test before starting. Budget your time wisely. Use the back of the pages if you need more space. You may tear the pages apart; we have a stapler at the front of the room.

It is a violation of the Academic Integrity Code to look at any exam other than your own, look at any reference material, or otherwise give or receive unauthorized help.

You will be expected to write Python code on this exam. We recommend that you draw vertical lines to make your indentation clear, as follows:

```
def foo():
    if something:
        do something
        do more things
    do something last
```

Unless you are explicitly directed otherwise, you may use anything you have learned in this course. You may use the backside of each page for extra room for your answers. However, if you do this, **please indicate clearly** on the page of the associated problem.

Question	Points	Score
1	2	
2	8	
3	14	
4	16	
5	14	
6	14	
7	17	
8	15	
Total:	100	

The Important First Question:

1. [2 points] Write your last name, first name, and netid at the top of each page.

Last Name: _____ First: _____ Netid: _____

References

String Operations

Expression	Description
`len(s)`	**Returns**: Number of characters in `s`; it can be 0.
`a in s`	**Returns**: True if the substring `a` is in `s`; False otherwise.
`s.find(s1)`	**Returns**: Index of FIRST occurrence of `s1` in `s` (-1 if `s1` is not in s).
`s.count(s1)`	**Returns**: Number of (non-overlapping) occurrences of `s1` in `s`.
`s.islower()`	**Returns**: True if `s` is *has at least one letter* and all letters are lower case; it returns False otherwise (e.g. `'a123'` is True but `'123'` is False).
`s.isupper()`	**Returns**: True if `s` is *has at least one letter* and all letters are upper case; it returns False otherwise (e.g. `'A123'` is True but `'123'` is False).
`s.lower()`	**Returns**: A copy of `s` but with all letters converted to lower case (so `'A1b'` becomes `'a1b'`).
`s.upper()`	**Returns**: A copy of `s` but with all letters converted to upper case (so `'A1b'` becomes `'A1B'`).
`s.isalpha()`	**Returns**: True if `s` is *not empty* and its elements are all letters; it returns False otherwise.
`s.isdigit()`	**Returns**: True if `s` is *not empty* and its elements are all numbers; it returns False otherwise.
`s.isalnum()`	**Returns**: True if `s` is *not empty* and its elements are all letters or numbers; it returns False otherwise.

List Operations

Expression	Description
`len(x)`	**Returns**: Number of elements in list `x`; it can be 0.
`y in x`	**Returns**: True if `y` is in list `x`; False otherwise.
`x.index(y)`	**Returns**: Index of FIRST occurrence of `y` in `x` (error if `y` is not in `x`).
`x.count(y)`	**Returns**: the number of times `y` appears in list `x`.
`x.append(y)`	Adds `y` to the end of list `x`.
`x.insert(i,y)`	Inserts `y` at position `i` in `x`. Elements after `i` are shifted to the right.
`x.remove(y)`	Removes first item from the list equal to `y`. (error if `y` is not in `x`).

Dictionary Operations

Expression	Description
`len(d)`	**Returns**: number of keys in dictionary `d`; it can be 0.
`y in d`	**Returns**: True if `y` is a key in dictionary `d`; False otherwise.
`d[k] = v`	Assigns value `v` to the key `k` in dictionary `d`.
`del d[k]`	Deletes the key `k` (and its value) from the dictionary `d`.
`d.clear()`	Removes all keys (and values) from the dictionary `d`.

Last Name: _____ First: _____ Netid: _____

2. [8 points total] **Short Answer**

   (a) [3 points] Name the four basic types in Python.

   The basic types are `int`, `float`, `bool` and `str`.

   (b) [3 points] What is the difference between `==` and `is`?

   The operator `is` compares the identifiers of the two objects/values. The operator `==` uses the `__eq__` of the object on the left to compare them (by default it is the same as `is`).

   (c) [2 points] In searching a list with n elements, approximately how many steps does linear search take? Approximately how many steps does binary search take?

   Linear search takes approximately n steps, while binary search takes approximately $\log(n)$ steps

3. [14 points total] **Testing and Exceptions**

   (a) [8 points] Consider the following function specification introduced in class.

```
def palindromish(s):
    """Returns True if s is a palindrome considering only letters (else False).

    The case of the letter and any non-letter characters are ignored.
    Example: palindromish('A man, a plan, a canal. Panama!') returns True.

    Precondition: s is a string"""
```

   Do not implement this function. Instead, provide a list of at least **five test cases** to test this function. For each test case provide: (1) the function input, (2) the expected output, and (3) an explanation of what makes this test *significantly* different.

   There are many possible answers. However, these were the main tests that we had in mind.

Input	Output	Reason
`'abc'`	False	Not a palindrome
`'aba'`	True	Simple palindrome
`'abA'`	True	Palindrome, ignoring case
`'a,ba'`	True	Palindrome, ignoring punctuation
`'ab,A'`	True	Palindrome, ignoring both
`''`	True	Empty string is a special palindrome

(b) [6 points] You have used the function `int` before. This function produces a `ValueError` if it cannot convert a string to an int (e.g. `int('a')`). In addition, it cannot convert a string with more than 4300 digits. In that case, assume it produces an `OverflowError` (which is *not* a subclass of `ValueError`). Using this knowledge, implement the function below. **You must enforce the precondition with the error specified**.

```
def nextint(s):
    """Returns the string representation of next int after string s

    If s does not represent an int, this function returns None. If it represents
    an int but has too many digits, this function returns the string 'inf'.
    Ex: nextint('10') returns '11', while nextint('a') returns None.

    Precond: s is a string; this func causes a TypeError if it is not."""
    # Enforce the precondition.
    if type(s) != str:
        raise TypeError()

    # Try block is if no errors occurred.
    try:
        n = int(s)
        n = n+1
        return str(n)
    except ValueError:
        return None
    except OverflowError:
        return 'inf'
```

4. [16 points] **Classes and Subclasses**

The very first step in Assignment 7 was to make a welcome message with the class `GLabel`. As you may recall, it has the following following primary attributes.

Attribute	Invariant	Description
`x`	`float`	x-coordinate of the label center.
`y`	`float`	y-coordinate of the label center.
`text`	`str`	The text to display.
`linecolor`	`a color value (see below)`	The color for the text.
`fillcolor`	`a color value (see below)`	The background color.

There are other attributes, such as `width` and `height`, but they are not important for this question. In particular, `width` and `height` are computed automatically from the `text`.

`GLabel` is often the superclass of more interesting classes. In this problem, you are to make a `GButton`, which provides a clickable button. We are not going to add mouse support, but we will add an attribute indicating whether or not it is pressed.

We want the button to change color when we press it. The simplest way to do this is to change the `fillcolor` to another value when it is pressed. This is shown to the right (the black button turns blue).

Last Name: _____ First: _____ Netid: _____

Implement this class below and on the next page. We have provided you with the specifications for the methods `__init__` and `draw`. You should fill in the missing details to meet these specifications. In addition, you must add the getters and setters (where appropriate) for the new attributes. **You are not allowed to add any attributes beyond those in the class invariant**. In addition, you should use `isinstance` instead of `type` when appropriate.

As part of this problem, you know that there are multiple ways to represent colors in Python (as strings, RGB objects, tuples, etc.). You can assume that you have the following function (**do not implement this function**).

```
def iscolor(value):
    """Returns True if value represents a valid GUI color
    """Precond: value can be anything"""
    # DO NOT IMPLEMENT THIS FUNCTION
```

```
from game2d import *

class GButton(GLabel):
    """A class representing a clickable button."""
    # MUTABLE ATTRIBUTES:
    # Invariant: _pressed is a boolean indicating if the button is pressed
    # Invariant: _visible is a boolean indicating if the button is visible
    # IMMUTABLE ATTRIBUTES:
    # Invariant: _downcolor is a color value for the button when pressed

    # DEFINE GETTERS/SETTERS AS APPROPRIATE. SPECIFICATIONS NOT NEEDED.
    def setPressed(self,value):
        """Sets whether the button is pressed

        Precond: value is a bool"""
        assert isinstance(value, bool)
        self._pressed = value

    def getPressed(self):
        """Returns whether the button is pressed"""
        return self._pressed

    def setVisible(self,value):
        """Sets whether the button is visible

        Precond: value is a bool"""
        assert isinstance(value, bool)
        self._visible = value

    def getVisible(self):
        """Returns whether the button is visible"""
        return self._visible

    def getDownColor(self):
        """Returns the button color when pressed"""
        return self._downcolor
```

Last Name: _____ First: _____ Netid: _____

```
# Class GButton (CONTINUED).


    def __init__( self,   x,   y,   text,   color,   shown = True ): # Fill in parameters
        """Initializes a button with the given text centered at (x,y).

        There are five parameters: x, y, text, color, and shown. The first three are the
        same as for GLabel. All buttons start with a fillcolor of 'black', and a linecolor
        of 'white'. The button is not pressed initially.

        The parameter color is the down color. The final parameter shown indicates
        if the button is visible. The parameter shown is OPTIONAL (default True).

        Precond: x and y are floats, text is a (nonempty) str, and  shown is a bool.
        The parameter color is a valid color (RGB or 3-element tuple of ints 0..255)."""
        assert isinstance(x, float)
        assert isinstance(y, float)
        assert isinstance(text, str) and text != ''
        assert isinstance(shown, bool)
        assert is_color(color)
        super().__init__(x=x,y=y,text=text,fillcolor='black',linecolor='white')
        self._pressed = False
        self._visible = shown
        self._downcolor = color




    def draw(                 self,      view                 ):     # Fill in parameters
        ""Draws this button to the provided view.

        If the button is not visible, it does not draw anything. Otherwise, it draws
        the button as normal if it is not pressed, and  uses the down color instead
        of the fillcolor if it is pressed.

        Precond: view is a GView object"""
        assert isinstance(view,GView)
        if self._visible:
            if self._pressed:
                oldcolor = self.fillcolor
                self.fillcolor = self._downcolor
                super().draw(view)
                self.fillcolor = oldcolor
            else:
                super().draw(view)
```

Last Name: _____ First: _____ Netid: _____

5. [14 points] **Nested Lists**

Implement the function below according to its specification. You do **not** need to enforce the precondition.

```
def rowfold(table):
    """Returns a new table whose rows are the sums of the rows in table.

    Each row in the new table is the sum of all rows up to AND INCLUDING
    that row in the original table.

    Examples:
        rowfold([[1],[2],[3],[4]]) returns [[1],[3],[6],[10]]
        rowfold([[1,2],[3,4],[5,6]]) returns [[1,2],[4,6],[9,12]]
        rowfold([[1,2,3],[4,5,6]]) returns [[1,2,3],[5,7,9]]
        rowfold([[1,2,3]]) returns [[1,2,3]]

    Precondition: table is a non-empty rectangular 2D list of integers"""
    # The new table should NOT share any rows with the old table
    # Create the accumulator
    result = []

    # Copy the first row
    firstrow = table[0][:]
    result.append(firstrow)

    # Sum the other rows
    for r in range(1,len(table)):
        row = []
        for c in range(len(table[0])):
            value = table[r][c]+result[r-1][c]
            row.append(value)
        result.append(row)

    return result
```

6. [14 points] **Recursion**

If you have ever downloaded DLC for a game, or redeemed a coupon online, you know that they are often defined as groups letters and numbers separated by dashes. The simplest variation has just one dash and groups its letters in numbers in blocks of four, like this: `K97J-FTRE`.

The function below takes a string of (potentially lower case) letters and numbers and returns it as a string of (upper case) letters and numbers separateds into groups of 4. If the length of the string is not divisible by 4, the leftovers at the end are dropped.

Implement this function using recursion. You do **not** need to enforce the precondition. **Answers using loops will receive no credit**.

```
def couponify(code):
    """Returns code properly formatted as a coupon code in blocks of 4.

    Each block of a coupon code is composed of upper case letters and numbers.
    Dashes are used to separate the blocks from each other.

    Example: couponify('k97J') returns 'K97J'
             couponify('K97j9876') returns 'K97J-9876'
             couponify('K97JfTRe9876') returns 'K97J-FTRE-9876'
             couponify('K97') returns ''
             couponify('K97J8') returns 'K97J'
             couponify('k97JfTRe9') returns 'K97J-FTRE'

    Preconditions: code is a (possibly empty) string of letters and numbers"""
    # Base case is anything 4 elements or less
    if len(code) < 4:
        return ''        # Drop anything too small
    elif len(code) == 4:
        return code.upper()  # Convert lower case letters

    # Break up into groups of 4
    left = code[:4]
    left = left.upper()  # Convert lower case letters
    right = couponify(code[4:])

    # Combine the result
    if len(right) > 0:
        return left+'-'+right
    else:
        return left
```

Last Name: _____ First: _____ Netid: _____

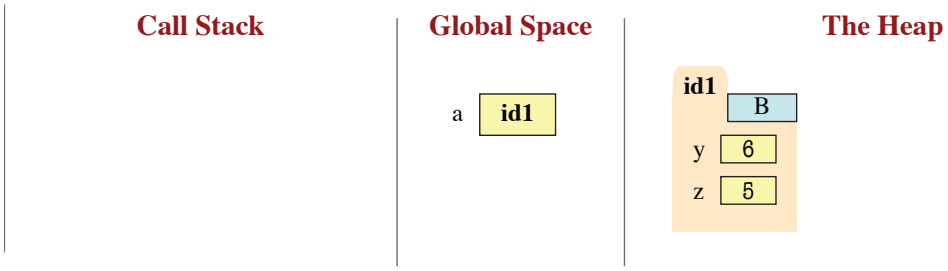
7. [17 points] **Call Frames**

Consider the two (undocumented) classes below, together with their line numbers.

```
1  class A(object):
2      x = 3
3
4      def __init__(self,x):
5          self.y = x
6
7      def f(self):
8          if self.x == 0:
9              return self.y
10         else:
11             return self.g(self.x-1)
12
13     def g(self,x):
14         return 2*self.f()
```

```
16 class B(A):
17     x = 1
18     y = 12
19
20     def __init__(self,x):
21         super().__init__(x+1)
22         self.z = x
23
24     def g(self,y):
25         self.x = y
26         return 3*self.f()
27
28     def h(self,y):
29         return y+2
```

If we create an object `a = B(5)`, we get the global variable and heap object shown below (**Do not diagram this constructor call**).



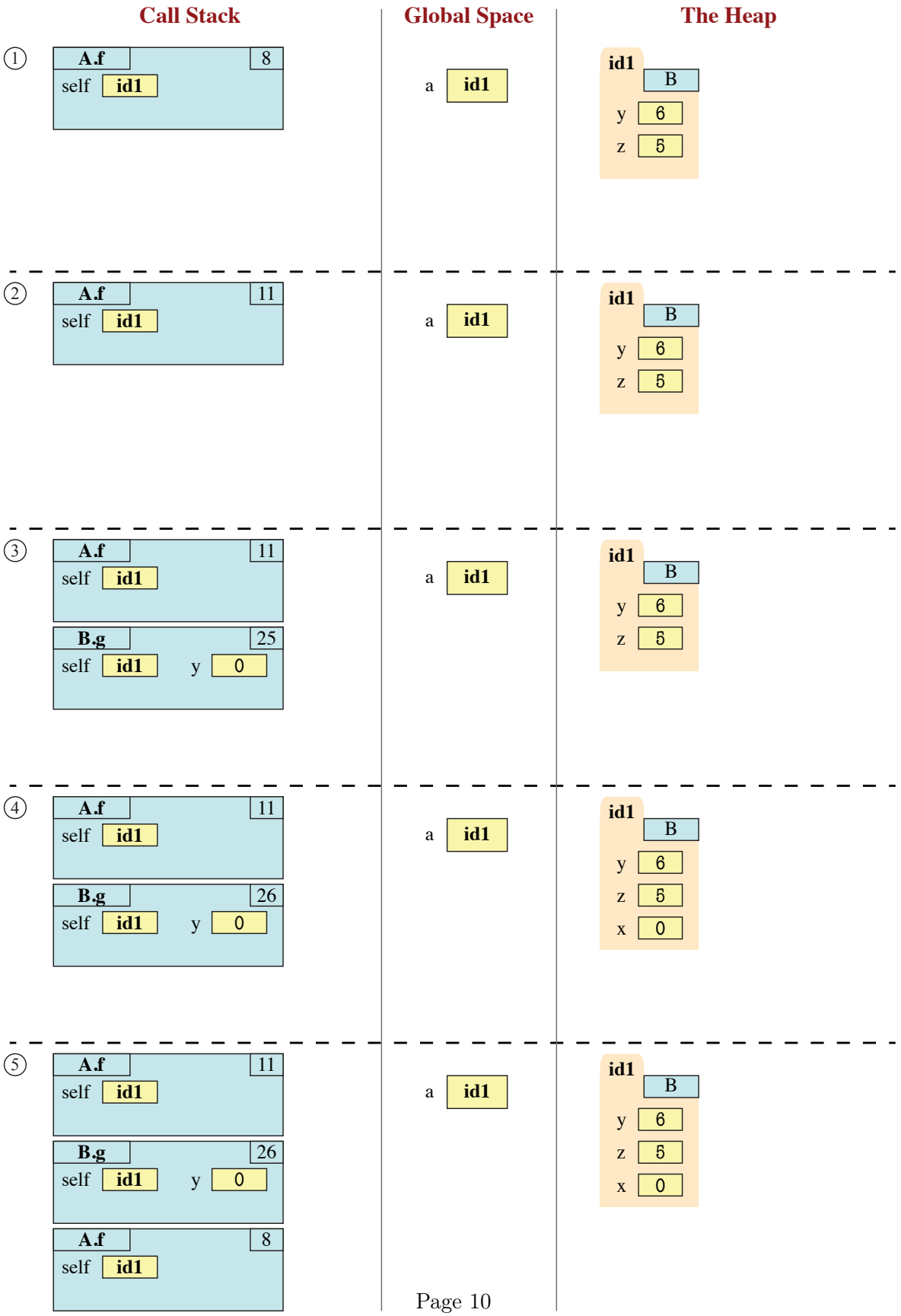
On the next two pages, diagram the evolution of the function call

```
b = a.f()
```

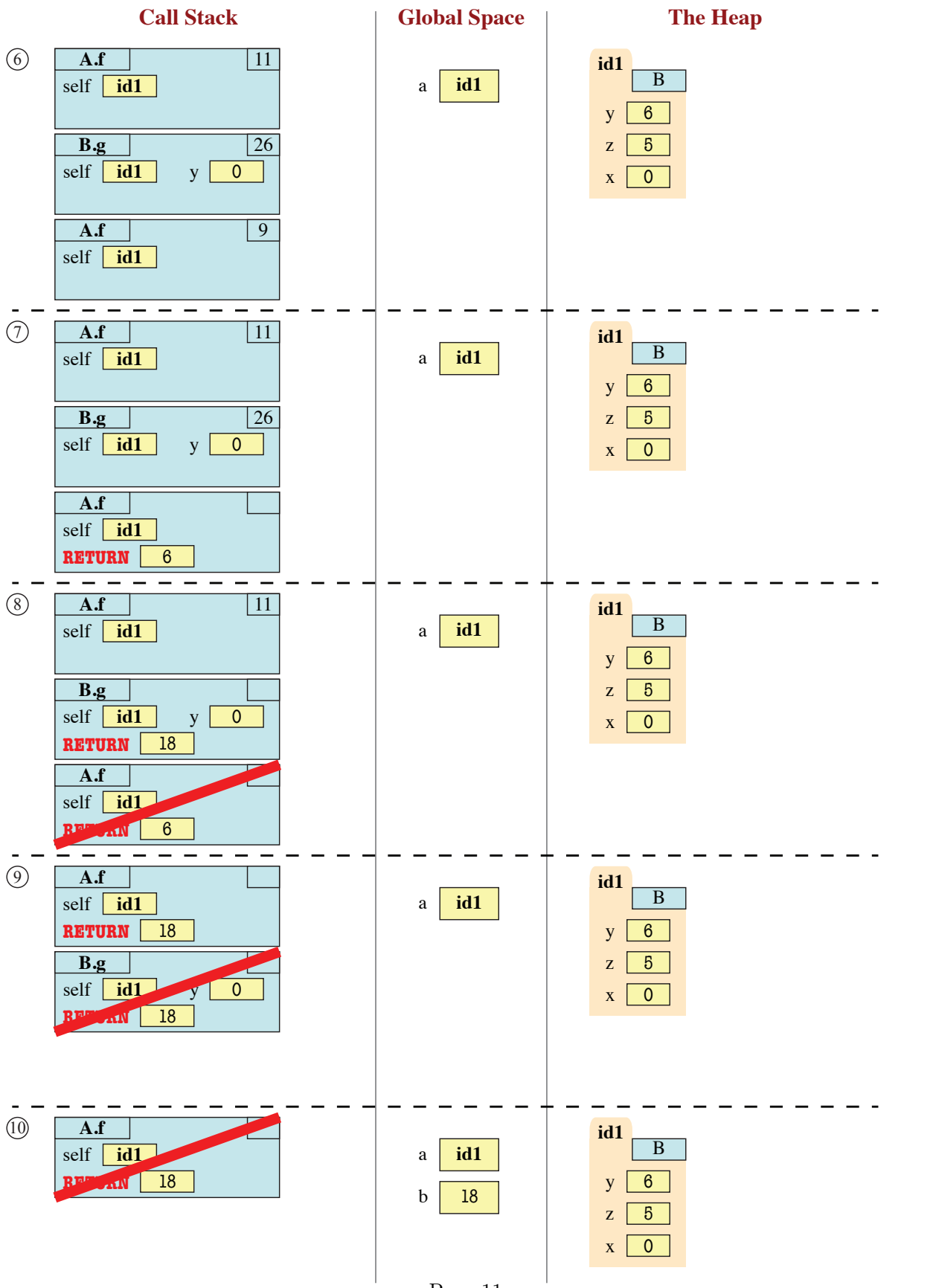
Diagram the state of the *entire call stack* for the this method call when it starts, for each line executed, and when the frame is erased. If any other functions are called, you should do this for them as well (at the appropriate time). This will require a total of **ten** diagrams.

You should draw also the state of global space and the heap at each step. You can ignore the folders for the function definitions and the class folders. **Only draw folders for objects (not classes)**. To help conserve time, you are allowed (and **encouraged**) to write "unchanged" if no changes were made to either a call frame, the global space, or the heap. In the case of the heap, you can point out that individual folder ids are unchanged.

Last Name: _____ First: _____ Netid: _____



Last Name: _____ First: _____ Netid: _____



8. [15 points total] **Generators**

Implement the generators below using anything learned in class. However, note the precondition of `emitruns`. Iterables that are not lists or strings cannot be sliced and have no length. **They also cannot be converted to lists**, as they may be infinite. Do **not** enforce the preconditions.

(a) [6 points]

```
def initials(s):
    """Generates the initials of s.

    An initial is any character at the start of a string or after a space.
    Ex: initials('John Smith') generates 'J' and 'S'
    Ex: initials('Ludwig von Mises') generates 'L', 'v', and 'M'
    Precond: s is a string"""

    # Track the previous character
    prev = ' '
    for ch in s:
        if prev == ' ':
            yield ch
        prev = ch
```

(b) [9 points]

```
def emitruns(stream):
    """Generates the run list of the given iterable.

    A run list starts at the beginning of the iterable, and stops when it reaches
    a number less than the latest value. Then it starts a new run list.
    Ex: emitruns([1,2,0,3,5,2,7]) generates [1,2], [0,3,5], and [2,7]
    Ex: emitruns([1,2,2]) generates [1,2,2] (only one run).

    Precond: stream is an iterable of ints"""

    # Need an accumulator for each run
    run = []

    for x in input:
        # Compare with the last element in the run
        if len(run) > 0 and x < result[-1]
            yield run
            run = [x]
        else:
            # Add it to the run
            run.append(x)

    # If anything left over, yield it
    if  len(run) > 0:
        yield run
```