Last Name: _____ First Name: _____ Cornell NetID, all caps: _____

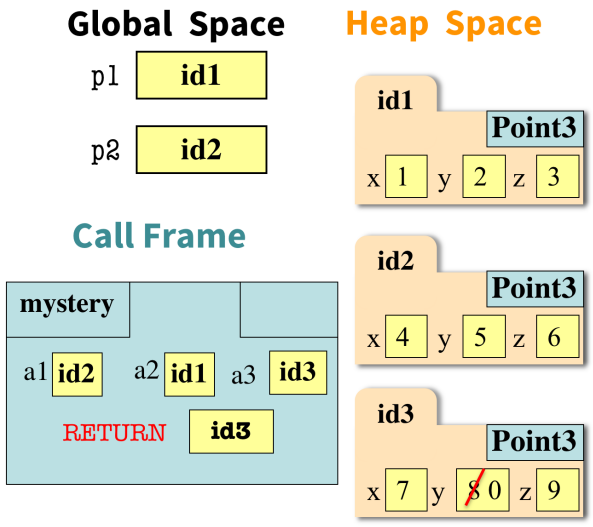
CS 1110 Regular Prelim 1 Solutions March 2018

Last Name: _____ First Name: _____ Cornell NetID: _____

1. [7 points] **What's the point?**

   Consider the Point3 class as it was defined in lecture, with 3 attributes: `x`, `y`, and `z`. Complete the code so that it will create the following memory diagram. Do not change any of the code provided, but (1) fill in the arguments to the incomplete call to `mystery` on line 18 and (2) complete the body of the function `mystery`. The third point on the Heap can be initialized with integers; it **does not** need to be a calculation involving the attributes of the first two points.

```
1  import shapes
2
3  p1 = shapes.Point3(1,2,3)
4  p2 = shapes.Point3(4,5,6)
5
6  def mystery(a1, a2):
7      # TASK #2 complete definition of mystery
8      # use as many lines as you need
9
10
11
12
13
14
15
16 # TASK #1: complete the call to mystery
17
18 mystery(    ,     )
```



Solution:
Task 2:

```
a3 = shapes.Point3(7,8,9)
a3.y = 0
return a3
```

Task 1:

```
mystery(p2, p1)
```

2. **Make the grade!** A talented but morally questionable CS 1111 student has written a script `make_my_grade()` that takes a list of lab grades, which is a list of ints, and changes them. First, every grade that is 0 is turned into a 5. Then all grades are doubled.

   (a) [6 points] Write **three conceptually distinct test cases** for function `make_my_grade()`.

   Test case #1
   Input and expected output:

   Rationale:

   Test case #2
   Input and expected output:

   Rationale:

   Test case #3
   Input and expected output:

   Rationale:

Solution:

```
input: [1,2,3,4]
output: [2,4,6,8]
test doubling for non-0 numbers

input: [0,0]
output: [10,10]
test function for 0's only

input: [1,0,2,0]
output: [2,10,4,10]
test function for mixed 0's and non-0's OR standard test case

input: []
output: []
edge case, empty list

input: [-1,2,-3]
output: [-2,4,-6]
test negative numbers

input: [1.0,0.0,-5.0]
output: [2.0,10.0,-10.0]
test floats

input: [0,10,20]
output: [10,20,40]
test numbers that contain the digit 0 that are not 0
```

(b) [7 points] Lab grades in CS 1111 are at most 10 points. Anything higher than 10 will raise a red flag and alert the staff that something fishy is going on, as will too many 10s.

Since you are also talented but morally questionable, you decide to write a helper function to be used by the previous script. It takes two lists (the original grades, `original`, and the doctored grades, `doctored`) and an index `i`.

If the element at index `i` in `doctored` is larger than 10, *and* there are no more than *three* 10s in `doctored`, then replace the element at index `i` in `doctored` with a 10.
Otherwise, replace the element of `doctored` at index `i` with the corresponding score from `original`.
Nothing is returned.

Implement the function as described, ignoring the need for any preconditions for now.

```
def check_grade(original, doctored, i):
```

Solution:

```
if doctored[i] > 10 and doctored.count(10) <= 3:
    doctored[i] = 10
else:
    doctored[i] = original[i]
```

(c) [2 points] Preconditions are often stated because if one violated them, the function would raise an error.

What is **one** precondition you should add to the specification of the function above? In other words, what condition (if violated) would cause your implementation to raise an error, and what kind of error would that be? (You do not have to give the exact name of the error; just describe it.)

Solution:
Valid index into both lists (IndexError)
Two lists of ints and/or floats (TypeError)
Lists of the same length (IndexError)
(OK if students cannot precisely name the Python error, but generally describe it.)

Last Name: _____ First Name: _____ Cornell NetID: _____

3. **Come on, get happy!**

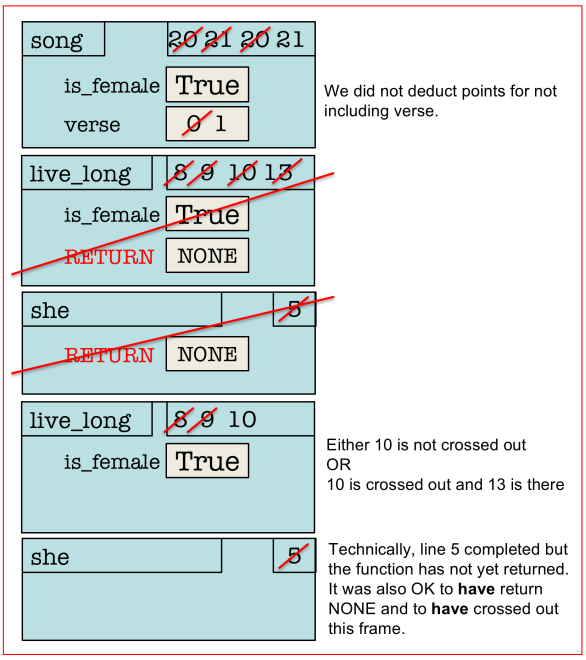
(a) [11 points] **What does the Call Stack look like?**

The Dutch version of "Happy Birthday" says "Long shall he/she live in glory". The first half of the sentence is repeated 3 times, then the second half of the song is repeated 3 times. Below on the left is the (error-free) code which prints out this Dutch song. On the right, draw the full call stack as it would look when **line 5 has just been executed** for the **second** time. Include crossed-out frames.

```
1  def he():
2      print("hij")
3
4  def she():
5      print("ze")
6
7  def live_long(is_female):
8      print("Lang zal")
9      if is_female:
10         she()
11     else:
12         he()
13     print("leven")
14
15 def in_glory():
16     print("In de gloria")
17
18 def song(is_female):
19     """ Happy Birthday in Dutch"""
20     for verse in list(range(3)):
21         live_long(is_female)
22     for verse in list(range(3)):
23         in_glory()
24
25 song(True)
```

Call Stack

Solution:



(b) [1 point] **What went wrong?**

Below on the left is code with one or more errors. Below on the right is the error message that is printed when the code is run. Fix the **one** line of code that led to the error message in the traceback. (**Fix only the error reported in this traceback.**)

```
1  def happy_holiday(holiday):
2      print("Happy "+holiday)
3
4  def dear():
5      print("Dear "+name)
6
7  def to_you():
8      print("to "+"you")
9
10 def line_with_name(name):
11     happy_birthday(name)
12     dear(name)
13
14 def basic_line(holiday):
15     happy_holiday(holiday)
16     to_you()
17
18 def song():
19    basic_line("Birthday")
20    basic_line("Birthday")
21    line_with_name("Teo")
22    basic_line(200)
23
24 song()
```

```
Traceback (most recent call last):
  File "happy_error.py", line 24, in <module>
    song()
  File "happy_error.py", line 21, in song
    line_with_name("Teo")
  File "happy_error.py", line 11, in line_with_name
    happy_birthday(name)
NameError: name 'happy_birthday' is not defined
```

Solution:
Change line 11 to call happy_holiday. (Not OK to change the function name in line 1, because line 14 calls it.)

4. [21 points] Complete `edit(instring, c1, c2)` below so that it obeys the following specification. (Don't include a docstring.)

   Preconditions: `instring` is a (possibly empty) string; `c1` and `c2` are two different strings, both of length 1.

   If `instring` does not contain any `c1`s, return `False`

   if `instring` contains 2 or more `c1`s,
   or contains exactly 1 `c1` but no `c2`,
   or contains exactly 1 `c1` and a `c2` before the `c1`, return the int `0`.

   Otherwise, return a version of `instring` where the text between `c1` and the first `c2` after it, inclusive, has been removed.

   Examples:

instring	c1	c2	what to return
"exam"	'('	')'	False
"B. (Bats) Wayne, M. (Prez)"	'('	')'	0
"(:"	'('	')'	0
"1) CS1110 (Intro)"	'('	')'	0
"T'Challa,(B.P.)x yz"	'('	')'	"T'Challa,x yz"
"a.().a"	'('	')'	"a..a"
"a () a"	'a'	'b'	0

```
def edit(instring, c1, c2):
```

Solution:
Two (similar) solutions:

```
def edit(instring, c1, c2):
# BEGIN REMOVE
    c1count = instring.count(c1)
    c2count = instring.count(c2)
    if c1count == 0:
        return False
    elif c1count > 1:
        return 0
    else:
        # c1count == 1
        c1_pos = instring.index(c1)
        c2_pos = instring.index(c2) # position of 1st c2
        if c2count == 0 or c2_pos < c1_pos:
            return 0

    #c1count === 1, there's an c2 after c1 but not before
    return instring[:c1_pos] + instring[c2_pos+1:]
```

```
def edit2(instring, c1, c2):
    # less readable
    c1count = instring.count(c1)
    c2count = instring.count(c2)

    if c1count == 0:
        return False
    # The order of these matters ...
    elif c1count > 1 or c2count == 0 or (instring.find(c2) < instring.find(c1)):
        return 0
    else:
        return instring[:instring.index(c1)] + instring[instring.find(c2)+1:]
```

5. [8 points] Recall from A2 that Player objects have an int attribute `holdings`. So, if `p` were a variable storing (the identifier of) a Player object, we could access the value of its `holdings` atttribute with the expression `p.holdings`.

   Suppose another class, `Team`, has been defined, where each Team object has an attribute `plist` that is a non-empty list of Player objects.

   Imagine someone asks us to write a function `switch(t1, t2, i1, i2)` with the following specification.

   > *Preconditions*: `t1` and `t2` are Teams. (They are allowed to be the same Team.)
   > `i1` is a valid index for `t1`'s list of Players, and `i2` is a valid index for `t2`'s list of Players.
   >
   > Swaps the `holdings` of the Player at index `i1` in `t1`'s list of players with the `holdings` of the Player at index `i2` of `t2`'s list of players. Does not return anything.

   Is it possible to write such a function?

   - If yes, write the body of the function below.
   - If not, explain why this is not possible, in 3-5 sentences, in the space below. You may use folder/call-frame diagrams in your explanation. *And*, write a line of code that would store in a variable `temp` the `holdings` of the Player at index 0 of the player list of Team `t1`.

   Hints: You may wish to draw object diagrams to make sure you understand the setup of the classes and lists involved. Try writing the function to decide whether or not it can be implemented.

```
def switch(t1, t2, i1, i2):
```

Solution:

```
    temp = t1.plist[i1].holdings
    t1.plist[i1].holdings = t2.plist[i2].holdings
    t2.plist[i2].holdings = temp
```

Alternately,

```
    temp = t2.plist[i2].holdings
    t2.plist[i2].holdings = t1.plist[i1].holdings
    t1.plist[i1].holdings = temp
```

6. [5 points] **A Lannister always pays his (or her) debts.**

   You are asked to complete a function (on the next page) that creates a payment plan for an amount owed. The amount owed is divided into a number of installments. After the first payment, a fee is assigned. The fee is a percentage of the installment amount and increases each month: no fee in the first month, 10% of the installment in the second month, 20% of the installment in the third month, and so on.

   Sample output for a few test cases are shown below:

```
>>> payment_plan(1000,1)
Each installment = $1000.0
#1: 1000.0
Total fees charged: $0.0

>>> payment_plan(1000,2)
Each installment = $500.0
#1: 500.0
#2: 550.0
Total fees charged: $50.0

>>> payment_plan(1000,3)
Each installment = $333.33
#1: 333.33
#2: 366.67
#3: 400.0
Total fees charged: $100.0

>>> payment_plan(1000,4)
Each installment = $250.0
#1: 250.0
#2: 275.0
#3: 300.0
#4: 325.0
Total fees charged: $150.0
```

   The `print` statements are already in the code and require you to create and give correct values to the variables `installment`, `curr_payment`, and `total_fees`.

```
1  def payment_plan(amount, num_payments):
2      """prints out a payment plan
3      amount: total amount owed (a float)
4      num_payments: # of installments (an int)
5
6      After the first payment, a fee is is assigned.
7      The fee is a percentage of the installment amount and increases each month:
8      - no fee in month 0
9      - 10% of the installment in month 1
10     - 20% of the installment in month 2, etc.
11
12     The function also prints out the total amount of fees one will have paid at the end
13     of the payment plan."""
14
15     # STUDENTS: initialize variable installment here with an assignment statement
16
17
18     print("Each installment = $"+str(round(installment,2)))
19
20     # STUDENTS: Initialize accumulator variable total_fees here with an assignment
21
22
23
24     # which_payment will have values 0 then 1 then 2...
25     # DO NOT CHANGE THE FOR-LOOP STRUCTURE GIVEN
26     for which_payment in list(range(num_payments)):
27         # STUDENTS: Compute curr_payment and create/update other variables as appropriate
28
29
30
31
32
33
34         print("#"+str(which_payment+1)+": "+str(round(curr_payment,2)))
35
36     print("Total fees charged: $"+str(total_fees))
```

Solution:
line 16:

```
installment = amount/num_payments
```

line 21:

```
total_fees = 0.0
```

lines 28 and following

```
        fee = installment * (0.1 * which_payment)
        curr_payment = installment + fee
        total_fees = total_fees + fee
```

---

7. [1 point] **Fill in your last name, first name, and Cornell NetID at the top of each page.**

   Solution:
   Always do this! It prevents disaster in cases where a staple fails.