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

CS 1110 Regular Prelim 1 Solutions March 2020

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

1. **Short Answer.** Write ERROR as shorthand for any error output.

(a) [4 points] What is printed out when the code below is executed?

```
alist = [20, 20]
count = 1
for a in alist:
    print(a)
    count = count * 2
print(count)
```

20
20
4

(b) [4 points] What is printed out when the code below is executed?

```
x = 1
y = 0
a = x >= 2 and (x/y) > 2
print("a is: " + str(a))
x = 16
b = x >= 2 and (x/y) > 2
print("b is:" + str(b))
```

a is: False
ERROR

(c) [4 points] What is printed out when the code below is executed?

```
def some_fun():
    print(i+6)
def more_fun(i):
    print(i-1)
i = 14
j = 10
some_fun()
more_fun(j)
```

20
9

(d) [4 points] Let `z` be a string containing at least one exclamation point. Write code that stores in variable `answer` the part of `z` that starts *just after* the first exclamation point in `z`.

One solution:

ep_pos = z.index('!')
answer = z[ep_pos+1:]

Alternately,
answer = z[z.index('!')+1:]
Other answers are also possible.

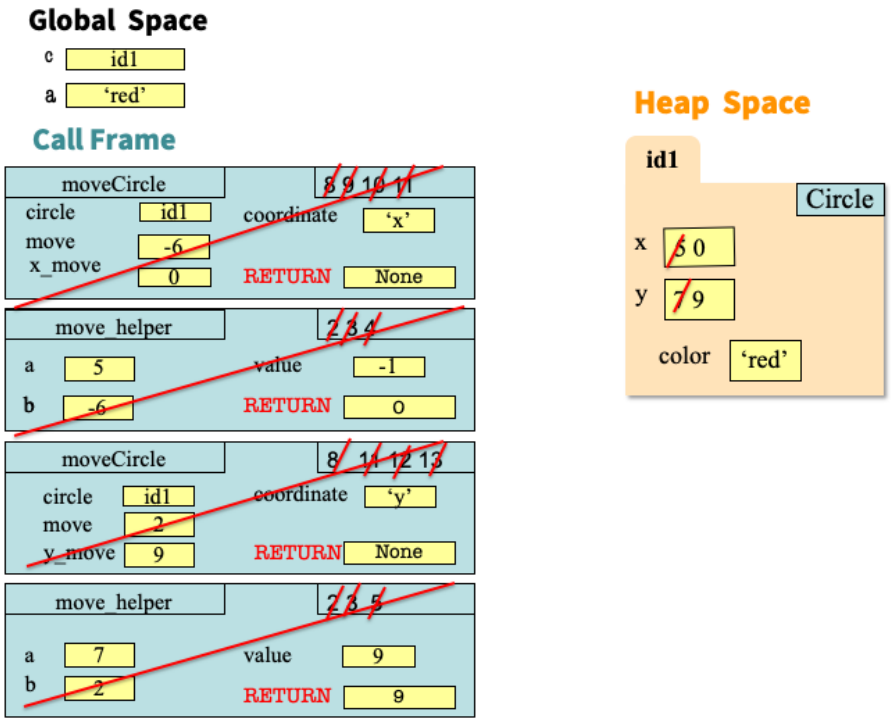
Last Name: _____ First Name: _____ Cornell NetID: _____

2. [26 points] `Circle` objects have three attributes: `x` [an `int`]: the x-coordinate of its center; `y` [an `int`]: the y-coordinate of its center; `color` [a non-empty `str`]: its color.

   A constructor expression like `Circle(5, 4, "blue")` creates a new `Circle` object with `x` attribute having value `5`, `y` attribute having value `4`, and `color` attribute having value `"blue"`.

```
1  def move_helper(a,b):
2      value = a+b
3      if value < 0:
4          return 0
5      return value
6
7  def moveCircle(circle, move, coordinate):
8      if coordinate == 'x':
9          x_move = move_helper(circle.x, move)
10         circle.x = x_move
11     else: # if executed, include line no. in frame
12         y_move = move_helper(circle.y, move)
13         circle.y = y_move
14
15 c = Circle(5,7,"red")
16 moveCircle(c,-6,'x')
17 moveCircle(c,2,'y')
18 a = c.color
```

Diagram the execution of lines 1-18 in the areas below.



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

3. **String Slicing**

(a) [8 points] A parenthetical phone number has parentheses around the first three digits (the area code), three more numbers, a hyphen, and then the last four numbers. So '(123)456-7890' is a valid parenthetical phone number.

Here is the specification for a function that judges whether a string is a valid parenthetical phone number.

```
def paren_phone_num(s):
    """Returns True if s is a valid parenthetical phone-number string,
    False otherwise.
    Precondition: s is a string.

    Example inputs and outputs:
    '(123)456-7890'     --> True
    '(123) 456-7890'    --> False
    '(123)456-7890-1'   --> False
    """
```

The above docstring gives some test cases, as inputs and expected outputs (omitting rationales). Write **four more distinct test cases**, as input and expected outputs (no need for `assert_equals` statements), plus rationale. Each test case needs to be conceptually distinct, for example, testing a different condition for a `False` rather than `True` return value.

For this problem, we want each test to have a different path through the conditionals in `paren_phone_num`. There are many different conditions on which a string could fail to be valid, and you should target your test cases to make sure any conditionals are being executed properly. Here are some sample cases we came up with:

Input	Output	Reason
`'1234567890'`	`False`	Phone number without any formatting
`'(123)456-'`	`False`	Substring of valid phone number
`'[123]456-7890'`	`False`	Not parentheses around area code
`'(123)456*7890'`	`False`	Not a dash between last two sections
`'(abc)456-7890'`	`False`	Area code is not a number
`'(123)xyz-7890'`	`False`	Middle section is not a number
`'(123)456-+$@!'`	`False`	Final section is not a number
`'(1t3)456-7890'`	`False`	Area code has both letters and numbers
`''`	`False`	Empty string

(b) [16 points] Now, implement the function.

You may not use for-loops in this function, only string operations and methods. You should *instead* use the string method `isdigit()`: for a string `x`, `x.isdigit()` returns `True` if all the characters in `x` are digits, `False` otherwise.

```
def paren_phone_num(s):
    """Returns True if s is a valid parenthetical phone-number string,
    False otherwise.
    Precondition: s is a string.

    Example inputs and outputs:
    '(123)456-7890'     --> True
    '(123) 456-7890'    --> False
    '(123)456-7890-1'   --> False
    """
    # Helpful position-numbering guide:
    # 0 1 2 3 4 5 6 7 8 9 10 11 12  <- possible indices
    # ( x x x ) x x x - x x  x  x   <- sample input template
```

Some solutions (other variants possible):

```
    # Check length and punctuation
    if len(s) != 13 or s[0] != '(' or s[4] != ')' or s[8] != '-':
        return False

    # Check the remaining stuff is numbers
    return s[1:4].isdigit() and s[5:8].isdigit() and s[9:].isdigit()


    # Alternate implementation of the above
    if not (s[1:4].isdigit() and s[5:8].isdigit() and s[9:].isdigit()):
        return False
    else:
        return True

    # Alternate implementation of the above
    if not s[1:4].isdigit() or not s[5:8].isdigit() or not s[9:].isdigit():
        return False
    else:
        return True
```

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

4. **Objects and Functions**

Consider a `Person` class with the attributes

- `name`: a string representing the name of this person
- `friends`: a (possibly empty) list of Person objects representing this person's friends

(a) [10 points] Implement the following function according to the specifications. Your implementation **must make effective use of `range()` in a for-loop.**

Hint: Recall the Python keyword `in`, which returns `True` if a value is in a sequence, and `False` otherwise. For example, `2 in [2, 3, 4]` evaluates to `True`, but `5 in [2, 3, 4]` evaluates to `False`.

```
def common(f1, f2):
    """Returns: a string list containing the names of the people that are in
    both Person list f1 and Person list f2.

    Example: Let p1, p2, ..., p6 be Person objects.  If f1 is the list
    [p2, p3, p5] and f2 is the list [p3, p4, p6, p5], then common(f1, f2)
    returns a list containing the names of p3 and p5 (not p3 and p5 themselves).

    Precondition: f1 and f2 are each a nonempty list of Person objects.
    """
    incommon= []
    for i in range(len(f1)):
        person= f1[i]
        if person in f2:
            incommon.append(person.name)
    return incommon
```

(b) [5 points] Implement function `mutual_friends` according to the specifications below. Your implementation must use function `common` from part (a) in a meaningful way. Assume `common` has been correctly implemented. Pay attention to the specifications of both `mutual_friends` and `common`.

```
def mutual_friends(p1, p2):
    """Returns: a string list containing the names of the mutual friends of
    Persons p1 and p2.  If p1 and p2 have no mutual friends, return an empty
    list.

    Precondition: p1 and p2 are each a Person object.
    """


    if p1.friends==[] or p2.friends==[]:
        return []
    return common(p1.friends, p2.friends)
```

(c) [9 points] Implement the following function according to the specifications below. Your implementation **must use a "for-each" loop meaningfully, i.e., you cannot use `range()` in your loop**.

```
def nickname_friends(p):
    """Returns: the number of names modified.  This function modifies
    Person p's friends list such that the names longer than 5 characters will
    will be truncated to the first 5 characters and a "u" is appended.  Names 5
    characters in length or shorter remain unchanged.

    Example: If p has 3 friends named "Jonathan", "Benji", and "Tristan", then
    their names will become "Jonatu", "Benji" (unchanged), and "Tristu",
    respectively, and the function returns 2.

    Precondition: p is a Person object with a nonempty friends list.
    """
    changes= 0
    for friend in p.friends:
        if len(friend.name) > 5:
            friend.name = friend.name[:5] + 'u'
            changes += 1
    return changes
```

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

5. **Testing and Debugging** The function `can_get_along` uses the birth years of two people to determine if they are compatible according to the logic of the Chinese zodiac. There are multiple bugs in the code below, potentially spread out across multiple functions. Read the specifications of each function carefully. On the next page, you will be asked to identify and fix the existing bugs.

```
1  def can_get_along(year1, name1, year2, name2):
2      """Prints out compatibility.
3      Years are ints, which convert to signs.
4      """
5      a1 = chinese_zodiac(year1)
6      print(name1 + " is " + \
7          proper_grammar(a1[0]) + a1 + '.')
8      a2 = chinese_zodiac(year2)
9      print(name2 + " is " + \
10         proper_grammar(a2[0]) + a2 + '.')
11     if compatible(a1,a2):
12         print('They are a good match!')
13     print('They are not a good match.')
14
15 def chinese_zodiac(year):
16     """Returns: sign (as str) of year (int)
17     """
18
19     zodiac = ['Rat', 'Ox', 'Tiger',
20               'Rabbit', 'Dragon', 'Snake',
21               'Horse', 'Sheep', 'Monkey',
22               'Chicken', 'Dog', 'Pig']
23
24     y = year - 4.0
25     en = zodiac[y % len(zodiac)]
26     return en
27
28 def compatible(z1,z2):
29     """Returns: True if z1 and z2 compatible,
30     False otherwise.
31     'Rat', 'Dragon', and 'Monkey' are compatible;
32     as are 'Ox', 'Snake', 'Rooster';
33     as are 'Tiger', 'Horse', 'Dog';
34     as are 'Rabbit', 'Goat', 'Pig'.
35     """
36     match = [['Rat', 'Dragon', 'Monkey'],
37     ['Ox', 'Snake', 'Rooster'],
38     ['Tiger','Horse','Dog'],
39     ['Rabbit', 'Goat','Pig']]
40
41     for i in range(len(match)):
42         if z1 in match[i] or z2 in match[i]:
43             return True
44     return False
45
```

```
46 def proper_grammar(first_letter):
47     """Returns: 'a ' or 'an ', depending on
48     first_letter, a string consisting of a
49     single capital letter.
50     """
51     if is_vowel(first_letter):
52         return "an "
53     return "a "
54
55 def is_vowel(x):
56     """Returns: True if 'x' is a vowel,
57     False otherwise.
58
59     Preconditions:
60         `x` [str]: a string with length 1.
61     """
62     vowels = 'AEIOU'
63     if vowels.find(x) < len(vowels):
64         return True
65     return False
```

(a) [4 points] **First Bug**: Consider the following call to `can_get_along` and the Python error it triggers.

```
>>> can_get_along(1996,'Suzie', 1997,'Ahmad')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "zodiac_friends.py", line 5, in can_get_along
    a1 = chinese_zodiac(year1)
  File "zodiac_friends.py", line 25, in chinese_zodiac
    en = zodiac[y % len(zodiac)]
TypeError: list indices must be integers or slices, not float
```

Below, explain where (line number) and why this error is triggered. And, fix the problem by writing below how the code should be rewritten.

The variable y on line 24 has type float but is used to index the list zodiac; list indices must be integers. Rewrite: `y = year-4`

(b) [4 points] **Second Bug**: After the first bug (above) is fixed, the call

```
>>> can_get_along(1996,'Suzie', 1997,'Ahmad')
```

should print out the following lines:

```
Suzie is a Rat.
Ahmad is an Ox.
[some other output]
```

Instead, it does the following.

```
>>> can_get_along(1996,'Suzie', 1997,'Ahmad')
Suzie is an Rat.
Ahmad is an Ox.
[some other output]
```

Below, explain where (line number) and why this error is triggered. And, fix the problem by writing below how the code should be rewritten.

The condition in the if on line 63 is always True. Rewrite: `if vowels.find(x) != -1:` or if `vowels.find(x) >= 0:`

(c) [8 points] **Third and Fourth Bugs**: Consider the following call to `can_get_along`

```
>>> can_get_along(1989,'Ji-woo', 1995,'Liam')
Ji-woo is a Snake.
Liam is a Pig.
They are a good match!
They are not a good match.
```

We guarantee that Ji-woo and Liam are years of the Snake and the Pig, respectively.

Below, explain where (line numbers) and why the two problems are triggered. And, fix the problems by writing below how the code should be rewritten.

Firstly 'Pig' and 'Snake' are not compatible, as can be seen from the docstring from compatible; but line 42 is just looking for whether `z1` or `z2` is in some list in `match`, so this will always be true.

Fix: `or` should be changed to `and` on line 42.

Secondly, `can_get_along` prints twice when `compatible` returns true because line 13 is always executed. Fix: line 13 should be converted to:

```
else:
    print('They are not a good match.')
```

6. [1 point] **Fill in your last name, first name, and Cornell NetID at the top of each page.**

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