Computer Science 1 — CSci 1100 — Spring 2017
Exam 3
April 17, 2017

Name: ____________________

RCS ID:									@rpi.edu

RIN#: _______________

Instructions:

- You have 90 minutes to complete this test.
- Clearly print your name, RCS ID (in all capital letters) and your RIN at the top of your exam.
- You must **have your Student ID** and you must be seated in the **correct section**. Failure may incur up to a **20 point penalty for each infraction.** If you are not sure if you are in the right section, please see a proctor before the exam starts.
- You may use only one double-sided crib sheet. Put your name and your RCS ID on it and turn it in at the end of the exam. Otherwise, put away all books, laptop computers, and electronic devices.
- Please read each question carefully several times before beginning to work.
- We generally will not answer questions except when there is a glaring mistake or ambiguity in the statement of a question.
- There are no Python syntax errors anywhere on this test.
- Unless otherwise stated, you may use any technique we have covered thus far in the semester to solve any problem.
- Please state clearly any assumptions that you have to make in interpreting a question.
- There are **five questions** on this test worth a total of **100 points**.
- When you are finished with this test please turn it into one of the proctors along with your crib sheet. After you show the proctor your student id you will be free to leave the exam room.

1. (**12 points**) Show the output of the following:

Code: | **Answer:**

```
# Part (a)
d = dict()
d[True] = []
d[False] = []
for c1 in range(2):
  for c2 in range(2):
      d[c1>c2].append('{}>{}'.format(c1,c2))
print("These are True:")
for trues in d[True]:
  print(trues)
print("These are False:")
for falses in d[False]:
  print(falses)
```

```
# Part (b)
S1 = set(["albatross", "bluebird", "crow", "duck"])
S2 = set(["bluebird", "duck", "falcon", "heron"])
print(S1 | S2)
print(S1.symmetric_difference(S2))
print(S1.union(S2) - S2.intersection(S1))
print(sorted(S2 - S1))
```

```
# Part (c)
class s(object):
    def __init__(self):
        self.d = [1.1, 2.2]
    def __str__(self):
        s = "{}: {}".format(len(self.d), self.d)
        return s
    def i(self, val):
        for ctr in range(len(self.d)):
            if self.d[ctr] >= val:
                return ctr
        return len(self.d)
    def a(self, val):
        if self.i(val) == len(self.d):
            self.d.append(val)
        elif self.d[self.i(val)] != val:
            self.d.insert(self.i(val), val)
obj = s()
print(str(obj))
obj.a(0.5)
print(str(obj))
obj.a(4)
print(str(obj))
obj.a(2.2)
print(str(obj))
```

2. (**12 points**) For each of the following write a **single line of Python code** to solve the problem. You can assume you are typing the command into the Python Shell. You must only use techniques we have covered thus far in the course. Any use of a `for` loop or a `while` loop will result in a 0 for that answer.

Code:	**Answer:**
(a) Given a list `L` containing duplicates, return a new list of all the values in L with duplicates removed. The order of entries in the returned list is not important. `d`. For example, given `L` `L = [1, 7, 9, 7, 1, 3, 6, 12, "apple", \` `"cat", "apple"]` your code should return `[1, 3, 6, 7, 'cat', 9, 12, 'apple']`	
(b) Given sets `s1`, `s2` and `s3`, create a new set `t` that contains all values that are in exactly two of the sets. For example, given `s1 = set(['train', 'car', 'bus', 'plane'])` `s2 = set([1, 3, 'car', 'train'])` `s3 = set([ 'bus', 'car', 3 ])` then after your code `t` should be `{'train', 3, 'bus'}` **Hint:** Start by writing an expression to print out all values that are in `s1` and in `s2` but not in `s3`.	
(c) Given a dictionary `d` whose keys and values are both numerical values (integers or floats), write an expression that evaluates to `True` if and only if the sum of the keys is larger than the sum of the values. As examples, the expression should evaluate to `True` for `d = {6: 7.1, 2:3, 10: 4}` and to `False` for `d = {1: 7.1, 2.2:3, 10: 4}`	

3. (**12 points**) Suppose you have a dictionary to represent your phone contacts. The keys are the names of people and the value associated with each key is the set of all that person's phone numbers, represented as strings (and only three digits for simplicity's sake). Here is an example:

```
contacts = {}
contacts['Bob'] = set(['100', '909'])
contacts['Alice'] = set(['505', '101'])
contacts['Chad'] = set(['999'])
contacts['Danielle'] = set(['123' ,'234', '345'])
```

   (a) (**6 points**) Write a function called `add_contact` that adds a phone number to a contact. If the contact does not exist, the function should create the contact. For example, the function call

```
add_contacts( contacts, 'Bob', '108')
```

   would add a third number to the contact set for `'Bob'`

(b) (**6 points**) Write a function `find_name` that takes the contact dictionary and a phone number and returns the name of the individual having that number. If the number is not in the dictionary, the function should return the string `"Unknown"`. For example, the call

```
find_name( contacts, '234' )
```

should return `'Danielle'`. You may assume each phone number is associated with at most one contact name.

4. (**32 points**) You are given a dictionary, `students` where the keys are student ids represented as six character strings and where the values are dictionaries with the following keys:

```
'NAME', 'COURSES', 'GRADES'
```

These refer to, in turn, the student's name (a string), the courses the student has taken (a list of strings), and the students grades (a list of strings 'A', 'B', 'C', 'D', or 'F').

```
s1 = { "NAME": "John Cleese", "COURSES": ["CSCI1100", "ENGR1100"], "GRADES": ["A", "B"]  }
s2 = { "NAME": "Michael Palin", "COURSES": ["CSCI2200", "BIOL2200", "CHEM1200"], "GRADES": ["A",
    "B", "A"]  }
s3 = { "NAME": "Eric Idle", "COURSES": ["CSCI1100", "BIOL2200"], "GRADES": ["C", "C"]  }
s4 = { "NAME": "Terry Gilliam", "COURSES": ["CSCI1100"], "GRADES": ["A"]  }
students = {"MP0001": s1, "MP0002": s2, "MP0003": s3, "MP0004": s4 }
```

(a) (**10 points**) Write a function called `count_courses` that takes the `students` dictionary as a parameter and returns the number of unique courses taken by the students. Your solution must use a set. For the above example version of `students`,

```
print( count_courses(students) )
```

should output

```
5
```

corresponding to `"CSCI1100", "ENGR1100", "CSCI2200", "BIOL2200", "CHEM1200"`

(b) (**10 points**) Write a function called `who_took` that accepts as an argument the `students` dictionary and generates a new dictionary where the courses are the keys and where a list of students who have taken the course are the values. As an example,

```
print( who_took(students))
```

should output

```
{'CSCI1100': ['Terry Gilliam', 'Eric Idle', 'John Cleese'], \
 'BIOL2200': ['Michael Palin', 'Eric Idle'], \
 'CSCI2200': ['Michael Palin'], \
 'CHEM1200': ['Michael Palin'], 'ENGR1100': ['John Cleese']}
```

(c) (**12 points**) Write code that assumes the `students` list is already created and that `who_took` is available. Repeatedly ask the user for a key until the user enters `"STOP"`. If the key is a course name, print the students who have taken that course. If the key is a student identifier, print the courses the student has taken. You can assume that no key can be both a course name and a student identifier. If the string is not `"STOP"` and is not a valid student ID or a valid course identifier, print `"No data"` and ignore it. You can assume that all keys and `"STOP"` are entered with correct capitalization. Here is an example from running the code:

```
Next key: MP0002
Student is Michael Palin
COURSES: ['CSCI2200', 'BIOL2200', 'CHEM1200']

Next key: MP0007
No data

Next key:             CSCI1100
STUDENTS: ['John Cleese', 'Terry Gilliam', 'Eric Idle']

Next key: STOP
```

5. (**32 points**) Consider the `Point2d` class we developed in lecture. For this question, we will develop a Point3d class. It will require 3 attributes for the `x`, `y`, and `z` coordinates, and 3 methods for the initializer, for the `__lt__`, and for a `dist` function.

   (a) (**8 points**) Write the Point3d class and the initializer. The initializer should take `x0`, `y0` and `z0` as initial values for the attributes and should specify the origin `(0, 0, 0)` as the default values if they are not provided. So, for example,

   ```
   p = Point3d()
   print("({}, {}, {})".format(p.x, p.y, p.z))
   ```

   should output

   ```
   (0.0, 0.0, 0.0)
   ```

   and

   ```
   p = Point3d(7.0, 2.0, -8.0)
   print("({}, {}, {})".format(p.x, p.y, p.z))
   ```

   should output

   ```
   (7.0, 2.0, -8.0)
   ```

(b) (**8 points**) Now write the distance function `dist` for the class. The function should return the distance of the point from the origin as `(x**2 + y**2 + z**2)**0.5`. You can assume your previous class definition and initializer exists and this method is just being added to the file. For example,

```
p = Point3d(7.0, 2.0, -8.0)
print("{}".format(p.dist()))
```

should output

```
10.816653826391969
```

(c) (**8 points**) Now write the `__lt__` function for the class. if `p` and `q` are two Point3d objects, `p < q` returns `True` if `p.dist() < q.dist()`. You can assume the methods you previously wrote exist and this method is just being added to the end of the file. For example,

```
p = Point3d(7.0, 2.0, -8.0)
q = Point3d(3.5, 1.0, -4.0)
print("p < q: {}".format(p < q))
print("q < p: {}".format(q < p))
```

should output

```
False
True
```

(d) (**8 points**) Finally, write a method `insertion_point` that takes a list of Point3d objects as a parameter. The `insertion_point` function should use the `__lt__` function to determine where a point should be inserted into a sorted list so as to maintain the sorted property. For example,

```
L = [Point3d(), Point3d(3.5, 1.0, -4.0), Point3d(7.0, 2.0, -8.0)]
p = Point3d(7.0, 1.0, -8.0)
print(p.insertion_point1(L))
```

should output

```
2
```

and

```
L = [Point3d(), Point3d(3.5, 1.0, -4.0), Point3d(7.0, 2.0, -8.0)]
p = Point3d(7.0, 2.0, -8.0)
print(p.insertion_point1(L))
```

should output

```
2
```

or

```
3
```

depending on if you decide to insert it before or after the original entry for `L[2]`