Ever so clearly print your email id:

Ever so clearly print your name:

True or False: I verified my solutions have been uploaded.

The number of questions I have asked or answered to the instructor during class is

- **None**
- **1 – 3**
- **4 – 6**
- **7 – 9**
- **10 or more**

Pledge:

Test rules

- Before you leave the room, check that you uploaded all thirteen of your solutions. Do not ask afterwards whether you can submit a forgotten solution.
- This pledged exam is closed notes. The only device you may access during the exam is your laptop.
- Any cheating can result in failing the class and the incident being referred to the Honor Committee.
- You may not access class examples, artifacts, solutions on the web, or your own past assignments during the test; that is, the only code you may access or view are ones that you develop for this test.
- The only windows allowed on your laptop are PyCharm and a single browser with tabs reachable from class website.
- Programs and modules should follow class programming practices; e.g., whitespace, identifier naming, and commenting if you think it is needed, etc.
- Whether a program is runnable or a module is testable is important.
- You must comment out or delete all debugging *print()* statements before submitting

Problem set

1. Implement program *whoami.py*. The program prints your University of Virginia email id and *nothing* else. For example, if your email id was mst3k, the program should produce the following output.

```
mst3k
```

2. Implement program *drops.py*. The program prompts its user for the number of gallons of interest. The program computes and then prints the number of drops of water for the indicated amount. There should be *no* other output. For your calculation, assume there are 90,922 drops of water per gallon. Your program should define a constant for number of drops of water per gallon.

Three possible program runs follow (FYI: a standard bathtub can hold eighty gallons of water, and 93701099000000000000000000000 is an estimate of the number of gallons of water in the Atlantic ocean)

```
Enter number of gallons: 1
90922
```

```
Enter number of gallons: 80
7273760
```

```
Enter number of gallons: 93701099000000000000000000000
8519491323278000000000000000000000
```

3. Implement program *mediate.py*. The program prompts its user for five strings. For the five strings, the program prints the one that comes in the middle alphabetically. There should be *no* other output.

   Three possible program runs follow.

```
Enter five strings: aaaa ddd b cc eee
cc
```

```
Enter five strings: fff hhh jjj iii ggg
hhh
```

```
Enter five strings: k l n o m
m
```

4. Implement program *asterisk.py*. The program prompts its user for integers. The program prints the product of the inputs. There should be *no* other output.

   Three possible program runs follow.

```
Enter integers: 3 1 4 1 5 9
540
```

```
Enter integers: 1 2 3 4 5
120
```

```
Enter integers: 2 4 6 8 10 12 14 16
10321920
```

5. Implement module *human.py*. The module defines a function *label*(). Function *label*() has one integer parameter *n.* Parameter *n* is an age of a person in years. The function returns a string classifying that age according to the following:

   - 0 – 17: Gen-Z
   - 18 – 34: Millennial
   - 35 – 50: Gen-X
   - 51 – 70: Boomer
   - 71 or greater: Gen-S

   The function already provides the following constant definitions.

```
LABEL_Z = "Gen-Z"           # between  0 - 17 years old
LABEL_M = "Millennial"      # between 18 - 34 years old
LABEL_X = "Gen-X"           # between 35 - 50 years old
LABEL_B = "Boomer"          # between 51 – 70 years old
LABEL_S = "Gen-S"           # 71 years old or greater
```

The built-in tester should produce the following output.

```
label( 17 ): Gen-Z
label( 33 ): Millennial
label( 48 ): Gen-X
label( 65 ): Boomer
label( 88 ): Gen-S
```

6. Implement module *wed.py*. The module defines a functions *ding*(). Function *ding*() has two list parameters *x* and *y*. The elements in *x* and *y* are all of the same type (e.g., all numeric or all strings). The function returns a new list whose elements are a sorted version of the union of *x* and *y*.

   The built-in tester makes use of the following definitions.

```
x1 = [ 2, 1, 1, 1 ];              x2 = [ 'p', 'i', 'n', 'k', 'y', ]
y1 = [ 1, 6, 2, 5, 4 ];           y2 = [ 't', 'o', 'e', ]
```

   The built-in tester should produce the following output.

```
ding( x1, y1 ): [1, 1, 1, 1, 2, 2, 4, 5, 6]
ding( x2, y2 ): ['e', 'i', 'k', 'n', 'o', 'p', 't', 'y']
```

7. Implement module *determine.py*. The module defines two functions *factorial*() and *combinations*().

   Function *factorial*() has a single integer parameter *n*. The function returns the *integer* product of the integer 1 through *n*; that is, 1 x 2 x 3 x ... x *n*. FYI: mathematicians denote this product as *n*!.

   The built-in tester should produce the following output.

```
5!: 120
6!: 720
10!: 3628800
```

   Function *combinations*() has two integer parameter *n* and *k*. The function returns as an *integer*, the number of possible combinations (ways) of choosing *k* things from a set of *n* things. The number of combinations equals

$$\frac{n!}{k! \cdot (n-k)!}$$

   The built-in tester should produce the following output.

```
combinations( 5 , 2 ): 10
combinations( 6 , 4 ): 15
combinations( 10 , 6 ): 210
```

8. Implement module *valuable.py*. The module defines two functions *count*() and *is_unique*().

Function *count*() has parameters *v* and *d*, where *v* is an arbitrary value and *d* is a dictionary. The function returns the number of keys in dictionary *d* whose value is *v*. Be aware that *d.values*(), the collection of values for *d*, is not a list. As such, there is no *count*() function for *d.values*(). However, a list version can be gotten with the expression *list*(*d.values*()).

The built-in tester makes use of the following definitions.

```
d1 = { 3: 1, 4: 1, 5: 9 }
d2 = { 'f': 'a', 'c': 'e', 'i': 'o', 'u': 's' }
```

The built-in tester should produce the following output.

```
count( d1, 1 ): 2
count( d1, 2 ): 0
count( d2, "a" ): 1
count( d2, "e" ): 1
```

Function *is_unique*() has a single dictionary parameter *d*. The function returns whether each of the values in *d.values*() occurs exactly once. If so, the function returns the logical literal *True*; otherwise, the function returns the logical literal *False*.

The built-in tester again makes use of the following definitions.

```
d1 = { 3: 1, 4: 1, 5: 9 }
d2 = { 'f': 'a', 'c': 'e', 'i': 'o', 'u': 's' }
```

The built-in tester should produce the following output.

```
is_unique( d1 ): False
is_unique( d2 ): True
```

9. Implement module `rolling.py`. The module defines a function *sixes*(). Function *sixes*() does not take any parameters. The function simulates the repeated rolling of a six-sided die whose sides are numbered 1 through 6. The rolling is repeated until a second 6 comes up. The function returns the list of rolls made including the second 6. Because the number of needed rolls is unknown, a *while* loop should be used.

   Important: the function does not use *random.seed*(). The seed is set by the built-in tester. The tester should produce the following output.

```
sixes(): [5, 4, 6, 3, 3, 2, 3, 1, 2, 1, 6]
sixes(): [2, 5, 1, 3, 1, 4, 4, 4, 6, 4, 2, 1, 4, 1, 4, 4, 5, 1, 6]
sixes(): [1, 1, 1, 3, 2, 6, 6]
```

10. Implement module `look.py`. The module defines two functions *composed*() and *which*().

   Function *composed*() has two string parameters *letters* and *word*. The function returns the logical literal *True* if the characters making up *word* are composed exclusively of the characters making up *letters* (repetition is allowed); otherwise, the function returns the logical literal *False*.

   The built-in tester should produce the following output.

```
composed( "arst", "rat" ): True
composed( "arst", "strata" ): True
composed( "cab", "bad" ): False
```

Function *which*() has a string parameter *letters* and a list of strings parameter *words*. The function returns which elements of the *words* are composed exclusively of the characters making up *letters*.

The built-in tester makes use of the following definition.

```
strings = ['bc', 'cc', 'bab', 'cad', 'bit', 'cab']
```

The built-in tester should produce the following output.

```
which( "abct", strings ): ['bc', 'cc', 'bab', 'cab']
```

11. Implement module *euclid.py*. The module defines three functions *pair*(), *series*(), and *perimeter*().

Function *pair*() has two coordinate parameters *p*1 and *p*2. The function returns the Euclidean distance between points *p*1 and *p*2. Suppose *p*1 equals (*x*1, *y*1) and *p*2 equals (*x*2, *y*2), then the Euclidean distance between *p*1 and *p*2 is

$$\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

FYI: math module function *math.sqrt*() should prove helpful. The built-in tester should produce the following output.

```
pair( (9, 4), (6, 1) ): 4.242640687119285
pair( (3, 5), (7, 5) ): 4.0
pair( (7, 2), (9, 2) ): 2.0
```

Function *series*() has a list parameter *spots* of coordinates. The function returns the distance traveling from *spots*[0] to *spots*[1], and from there to *spots*[2], and from there to *spots*[3], ... and ending up at *spots*[*n*-1], where *n* is the length of *spots*.

The built-in tester makes use of the following definition.

```
coordinates = [(9, 4), (6, 1), (3, 5), (7, 5), (7, 2), (9, 2)]
```

The built-in tester should produce the following output.

```
series( coordinates ): 18.242640687119284
```

Function *perimeter*() also has a list parameter *spots* of coordinates. The function returns the distance traveling from *spots*[0] to *spots*[1], and from there to *spots*[2], and from there to *spots*[3], ... from there to *spots*[*n*-1], and from there back to *spots*[0], where *n* is the length of *spots*.

The built-in tester also makes use of the following definition.

```
coordinates = [(9, 4), (6, 1), (3, 5), (7, 5), (7, 2), (9, 2)]
```

The built-in tester should produce the following output.

```
perimeter( coordinates ): 20.242640687119284
```

12. Implement module *datable.py*. The module defines two functions *flatten*() and *similar*().

Function *flatten*() has a single dataset parameter *d*. The function returns a simple list whose elements are the elements of first row of dataset *d,* followed by the elements of the second row of the dataset, followed by the elements of the third row of the dataset, and so on.

The built-in tester makes use of the following definitions.

```
d1 = [ [31, 28, 31, 30], [31, 30, 31, 31], [30, 31, 30, 31 ] ]
d2 = [ [1, 1, 2, 3], [5, 8, 13, 21], [34, 55, 89, 144], [233, 377, 610 ] ]
```

The built-in tester should produce the following output.

```
flatten( d1 ): [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
flatten( d2 ): [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Function *similar*() has two dataset parameters *d*1 and *d*2. The function returns whether datasets *d*1 and *d*2 have equivalent flattenings. If the flattenings are the same, the function returns the logical literal *True*; otherwise, the function returns the logical literal *False*.

The built-in tester makes use of the following definitions.

```
a1 = [[31, 28, 31, 30], [31, 30, 31, 31], [30, 31, 30, 31 ]]
b1 = [[31, 28, 31, 30, 31, 30, 31, 31, 30], [31, 30, 31 ]]

a2 = [[1, 1, 2, 3], [5, 8, 13, 21], [34, 55, 89, 144], [233, 377, 610, 987 ]]
b2 = [[1, 2, 3, 4], [5, 8, 13], [34, 55, 89], [233, 377, 610, 987 ]]
```

The built-in tester should produce the following output.

```
similar( a1, b1 ): True
similar( a2, b2 ): False
```

13. Implement module `pixie.py`. The module defines two functions *max_rgb*() and *tricolor*().

Function *max_rgb*() has a single parameter *pixel*, which is a 3-tuple with values between 0 and 255. Suppose *pixel* equals (*r, g, b)*. If *r* equals the max of the three values, the function returns (*r*, 0, 0). If instead *g* equals the max of the three values, the function returns (0, *g*, 0). Otherwise, the function returns (0, 0, *b*).

The built-in tester runs three tests using pixel colors

```
p1 = (80,75,70);      p2 = (80,95,70);      p3 = (80,75,99)
```

The tester should produce the following output.

```
max_rgb( p1 ): (80, 0, 0)
max_rgb( p2 ): (0, 95, 0)
max_rgb( p3 ): (0, 0, 99)
```

Function *tricolor*() has a single parameter *original*, which is a *PIL Image*. The function returns a new image with the same dimensions as *original*. The pixels in the new image are variations of the corresponding pixels in *original*. The variation is gotten using function *max_rgb*().

The built-in tester uses an image looking like www.cs.virginia.edu/~cs1112/images/pre-tricolor.png as the original, and shows an image equal to the one at www.cs.virginia.edu/~cs1112/images/post-tricolor.png. The true coloring of the two images are on the class web page.