COMMENTS

- You have 90 minutes to complete. People with accommodations have 90 minutes times their multiplier: 135 minutes (1.5) and 180 (2.0).
- By submitting solutions for this test, you are agreeing that you neither given nor received aid directly or indirectly to or from another test taker.
- By submitting solutions for this test, you are agreeing that you did not use directly or indirectly use materials from non-allowed sources.
- Check that you uploaded all your solutions. Do not ask later to submit a forgotten solution.
- The only device you may access during the exam is your laptop. The only open windows allowed are PyCharm and a browser with tabs linked from the class website.
- No outside help is permitted.
- The only code you may access are ones that you develop for this test.
- You may not access class notes, epistles, examples, artifacts, solutions on the web, or your own past assignments during the test.
- Code should follow class programming practices; e.g., whitespace, identifier naming, etc.
- None of your code should print or request input.
- Whether code is testable is important. Comment out or delete all debugging `print()` statements before submitting.
- Make sure all functions have at least one non-commented statement

1. Complete the implementation of *duck.py*. The module defines a function `quack()` with no parameters. The function returns the number of ducks – virtual or real – you earned this semester. The definition as written is:

```
def quack() :
    nbr_of_ducks_earned = 0     # replace 0 with number of ducks earned

    return nbr_of_ducks_earned
```

Unless you update the function definition, the built-in tester for the module produces the following.

```
quack(): 0
```

2. Implement module *mm.py*. The module defines a function `conv()` with one decimal parameter p, where p is a desired weight in pounds.

The function returns the integer number of marshmallows needed to make p pounds. To assist you, the module already defines a constant `WEIGHT_OF_ONE_MARSHMALLOW`, which is the weight of a typical marshmallow in pounds.

```
WEIGHT_OF_ONE_MARSHMALLOW = 0.0154324
```

The function return value should be gotten by *rounding* the decimal result of p divided by the weight of one marshmallow.

The built-in tester for the module should produce the following.

```
conv( 0.07716200000000001 ): 5
conv( 1 ): 65
conv( 2000 ): 129597
```

3. Implement module *com.py*. The module defines a function `sob()` with two integer parameters n and c. The function returns the integer number of ways w of choosing c elements from a list with n elements.

The formula for determining the number of ways w is

$$w = (x // (y \cdot z))$$

where

$$x = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

$$y = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot c$$

$$z = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n - c)$$

The built-in tester for the module should produce the following.

```
sob( 5, 3 ): 10
sob( 8, 2 ): 28
sob( 12, 4 ): 495
```

4. Implement module *tog.py*. The module defines a function `thob()` with two string parameters `s1` and `s2`, and a string list parameter `s3`. The function returns a new list of strings. The elements of the new list are those elements of `s3` that have both `s1` and `s2` as substrings.

The built-in tester makes use of the following string lists.

```
strings1 = [ "tango", "apple", "banana", "manna", "nada" ]

strings2 = [ "000", "001", "010", "011", "100", "101", "110", "111"]

strings3 = [ ]
```

The built-in tester for the module should produce the following.

```
thob( 'an', 'na', strings1 ): ['banana', 'manna']
thob( '0',  '11', strings2 ): ['011', '110']
thob( '0',    '',  strings3 ): []
```

5. Implement module *can.py*. The module defines a function `cmp()` with an integer list parameter x. The function returns a new list of integers. The first value in the new list is either −1, 0, or 1 depending on whether the first element of x is negative, zero, or positive, the second value in the new list is either −1, 0, or 1 depending on whether the second element of x is negative, zero, or positive, and so on.

The built-in tester for the module makes use of the following datasets.

```
x1 = [ 4, −5,  2, −5,  9 ]
x2 = [ 0,  8,  0, −7,  4, 4, 0 ]
x3 = [ 6,  8,  0,  2, 4 ]
```

The built-in tester for the module should produce the following.

```
cmp( x1 ): [1, −1, 1, −1, 1]
cmp( x2 ): [0, 1, 0, −1, 1, 1, 0]
cmp( x3 ): [1, 1, 0, 1, 1]]
```

6. Implement module *wid.py*. The module defines a function `get()` with two list parameters x and y. You can assume x and y have the same length. The function returns a new dictionary. In the new dictionary, the value at index 0 of x maps to the value at index 0 of y, the value at index 1 of x maps to the value at index 1 of y, and so on.

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

```
x1 = [ 'a', 'b', 'c', 'd', 'e' ]
y1 = [ 1, 2, 3, 4, 5 ]

x2 = [ 3, 1, 4, 1 ]
y2 = [ 'odd', 'odd', 'even', 'odd' ]
```

The built-in tester for the module should produce the following mappings (be aware your ordering of the mappings could be different).

```
get( x1, y1 ): {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
get( x2, y2 ): {3: 'odd', 1: 'odd', 4: 'even'}
```

7. Implement module *cre.py*. The module defines a function `tea()` with a parameter `v` and two integer parameters `r`, and `c`. The function returns a new data set with `r` rows and with each row having `c` columns. All values in the new data set are set to `v`.

The built-in tester for the module should produce the following.

```
tea(    0, 2, 5 ) : [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
tea(  'a', 3, 2 ) : [['a', 'a'], ['a', 'a'], ['a', 'a']]
tea( True, 4, 0 ) : [[], [], [], []]
```

8. Implement module *atad.py*. The module defines a function `nbr()` with a parameter `v` and a dataset parameter `d`. The function returns a new list of integers. The first value in the new list is the number of occurrences of v in the first row of the dataset, the second value in the new list is the number of occurrences of v in the second row of the dataset, and so on.

The built-in tester for the module makes use of the following datasets.

```
d1 = [ [ 3, 1, 4, 1, 5, 9, 2 ],
       [ 6, 5, 3, 5, 8, 9, 7, 9, 3, 2 ],
       [ 3, 8, 4, 6, 2, 6, 4, 3, 3 ],
       [ 8, 3, 2, 7, 9, 5, 0 ] ]

d2 = [ [ 2, 8, 8, 4, 1 ],
       [ 9, 7, 1, 6, 9, 3, 9, 9 ],
       [ 3, 7, 5, 1, 0, 5, 8, 2, 0, 9, 7, 4, 9 ] ]

d3 = [ [ 4, 4 ],
       [ ],
       [ 5, 9, 2, 3, 0, 7, 8, 1, 6, 4 ]]
```

The built-in tester for the module should produce the following.

```
nbr( d1, 9 ): [1, 2, 0, 1]
nbr( d2, 8 ): [2, 0, 1]
nbr( d3, 5 ): [0, 0, 1]
```

9. Implement module *spin.py*. The module defines a function `color()` with a single pixel parameter p. The function returns a new pixel whose R value is the maximum of p's RGB values, whose G value is the integer average of p's RGB values, and whose B value is the minimum of p's RGB values. The built-in tester for the module should produce the following.

```
spin( (50, 100, 200) ): (200, 116, 50)
spin( (241, 59, 136) ): (241, 145, 59)
spin( (90, 109, 80) ): (109, 93, 80)
```