COMPUTING ID _____ **PRINT NAME LEGIBLY:** _____

Read this entire page. You are responsible knowing what it says.

Honor code

- By submitting solutions for this test, you are agreeing that
  - You neither given nor received help directly or indirectly to or from anyone else.
  - You did not directly or indirectly use materials from non-allowed sources.

Important

- You must use our files when coding.
- None of your code should get input or produce output.
- Do not access past code or algorithms (yours, ours, or anyone else's).
- Do not access class notes, epistles, examples, artifacts, solutions on the web, or your own past assignments.
- Class personnel cannot help you debug your answers.
- Comment out or delete all debugging statements before submitting.
- Whether code is testable is important.
- 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.
- During the test you can access the course module descriptions and the course Python information sheet.
- You are responsible for submitting for your work, so check before leaving the test. Do not submit once the test is over.
- Code should follow class programming practices, e.g., whitespace, identifier naming, etc.
- Because the solutions are all short, commenting is not necessary.
- You might add comments if you were unable to complete a problem and want to explain what you were attempting to do.

Programming

1. Implement module *iam.py*. The module defines a single function `honorable()`. The function has no parameters. If you completed the test in a completely honorable manner, the function should return the lower-case string `"yes"`; otherwise, the function should return the string `"no"`.

   The expected output of the built-in tester is the following.

```
honorable(): yes
```

2. Implement module *calc.py*. The module defines a function btog(). The function has a single numeric parameter named b. The value of the parameter specifies the number of barrels of interest. The function returns the equivalent number of gallons. Note, there are 42 gallons per barrel.

   The built-in tester should produce the following output.

```
btog( 2 ): 84
btog( 3 ): 126
btog( 5 ): 210
```

3. Implement module *accum.py*. The module defines a function ulate(). The function has a single numeric list parameter named items. The function accumulates a total based on the elements of items. If an element of items is negative it is added to the accumulation. If an element of items is positive, the element is subtracted from the accumulation.

   The built-in tester defines three lists.

```
test1 = [ 2, -10, 2, -6, -12 ]
test2 = [ -15, 0, 13, -5, -5, 6, -14, 10, -7, -5, 10 ]
test3 = [ 11, 10, -7, 12, -8, -8, 10, -6, 15 ]
```

   The built-in tester should produce the following output.

```
ulate( test1 ): -32
ulate( test2 ): -90
ulate( test3 ): -87
```

4. Implement module *counts.py*. The module defines a function cmp(). The function has a single numeric list parameter named items. The function returns an integer list with three elements. The first element is the number of negative elements in items; the second element is the number of zero-valued elements in items; and the third element is the number of positive elements in items.

   The built-in tester defines three lists.

```
test1 = [ 2, -10, 2, -6, -12 ]
test2 = [ -15, 0, 13, -5, -5, 6, -14, 10, -7, -5, 10 ]
test3 = [ 11, 10, -7, 12, -8, -8, 10, -6, 15 ]
```

   The built-in tester should produce the following output.

```
cmp( test1 ): [3, 0, 2]
cmp( test2 ): [6, 1, 4]
cmp( test3 ): [4, 0, 5]
```

5. Implement module *dracula.py*. The module defines a function build(). The function has a single list parameter named items. The function returns a new dictionary. The keys of the new dictionary are the elements of items. In the new dictionary, an element from items maps to the number of the times the element is in items.

The built-in tester defines three lists.

```
test1 = [ 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 ]
test2 = [ ]
test3 = [ 'p', 'e', 'o', 'p', 'l', 'e' ]
```

The built-in tester should produce the following output.

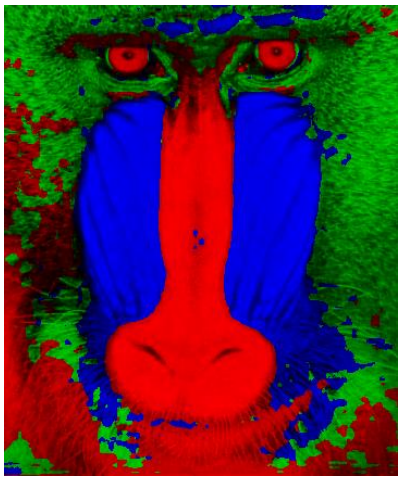
```
build( test1 ): {3: 2, 1: 2, 4: 1, 5: 3, 9: 1, 2: 1, 6: 1}
build( test2 ): {}
build( test3 ): {'p': 2, 'e': 2, 'o': 1, 'l': 1}]
```

Note: because Python dictionaries are not ordered lists. It is possible the tester output of the mappings in your dictionary can appear in a different order.

6. Implement module *slide.py*. The module defines a function `skew()`. The function has a single pixel parameter named `p`. The function returns a new pixel. Suppose `r`, `g`, and `b` are respectively, the RGB components of `p`.

   - If `r` equals the maximum of the RGB values of `p`, the function returns (`r`, 0, 0).
   - Otherwise, if `g` equals the maximum of the RGB values of `p`, the function returns (0, `g`, 0).
   - Otherwise, the function returns (0, 0, `b`).

   The built-in tester should produce the following image.



7. Implement module *sub.py*. The module defines a function `tract()`. The function has two strings parameters named `s1` and `s2`. The function returns a new string. The new string is composed of one occurrence of each character in `s1` that is not part of `s2`. The built-in tester should produce the following output.

```
tract( banana, apple ): bn
tract( explanation, planet ): xio
tract( kiwi, orange ): kiw
```