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

Honor

- 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.**
- The `WWHAD` strategy (what would a human do strategy) should serve you well.
- During the test you may not access past code or algorithms (yours, ours, or anyone else's).
- During the test you may not access class notes, epistles, examples, artifacts, solutions on the web, or your own past assignments during the test.
- Class personnel cannot help you debug your answers.
- All functions make use of tester module *etester.py*.
- None of your functions should modify any list or dataset parameters.
- None of your functions should print or get input.
- Comment out or delete all debugging `print()` statements before submitting.
- Whether code is testable is important. Every function needs to have at least one uncommented statement.
- None of the testing code should be modified.
- 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 exiting the testing. Late submissions will not be graded, so do not submit once your testing time is up.
- Code should follow class programming practices; e.g., whitespace, identifier naming, etc.
- Because the problems are 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.

Problems

1. Implement a ***program*** *diff.py*. The program does not define any functions. The program prompts its user for text. The program prints either literal `True` or `False` depending whether the words in the text are all different. Word comparison ignores case; e.g., "Love" and "love" are the same.

   Some possible program runs are

   ```
   Enter words: love is all you need
   True
   ```

   ```
   Enter words: a b c B
   False
   ```

   ```
   Enter words: A b a
   False
   ```

2. Implement module *verify.py.* The module defines a function *bi*() with a dictionary parameter *d*.

   Function *bi*() returns one of logical literals `True` or `False` depending whether dictionary *d* is a bijection. A dictionary is a *bijection* if no two keys in the dictionary map to the same value; that is, every key maps to a distinct value.

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

   ```
   d1 = { 0: "zero", 1: 2, "a": "b", True: False }
   d2 = { 1: 2, "a": "vowel", "e": vowel }
   d3 = { 1: 1 }
   d4 = { }
   ```

   And should produce the following output.

   ```
   bi( d1 ): True
   bi( d2 ): False
   bi( d3 ): True
   bi( d4 ): True
   ```

3. Implement module *fun.py*. The module defines a function *anagram*(). Function *anagram*() has two string parameters *s* and *t*.

   Function *anagram*() returns whether *s* and *t* are anagrams of each other. String *s* is an *anagram* of string *t,* if *s* is a rearrangement of the letters of *t*.

   The built-in tester should produce the following output.

   ```
   anagram( listen, silent ): True
   anagram( love, evil ): False
   ```

4. Implement module *factoring.py*. The module defines a function *common*(). Function *common*() has two integer parameters *x* and *y*.

   The function returns a list of the proper factors common to both *x* and *y*.

   The *factors* of a number *n* are those integers that produce a remainder of zero when they divide into *n* (e.g., 5 is a factor of 10 because 10 divided by 2 has remainder 0, 3 is not a factor of 10 because 10 divided by 3 has remainder 1).

   A factor of *n* is a *proper factor* if it is neither 1 nor *n* (e.g., the proper factors of 10 are 2 and 5).

   The built-in tester should produce the following output.

```
common( 2, 4 ) = []
common( 4, 4 ) = [2]
common( 18, 6 ) = [2, 3]
common( 40, 80 ) = [2, 4, 5, 8, 10, 20]
common( 90, 100 ) = [2, 5, 10]
```

5. Implement module *rolling.py*. The module defines a function *six*(). Function *six*() does not take any parameters. The function simulates the repeated rolling of a six-sided die whose sides are numbered 1 through 6. The simulated 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 needs to be used. Important: the function does not use *random.seed*() (the seed is set by the built-in tester).

   The built-in 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]
```

6. Implement module *nucleotide.py*. There are four kinds of nucleotide molecules – adenine, cytosine, guanine, and thymine. They are commonly referred to respectively as A, C, G, and T. Nucleotides A and T are *complementary* molecules, nucleotides C and G are also *complementary* molecules.

   The module defines functions *extract*(), *snip*(), and *complement*(). They all deal with nucleotides, where nucleotides are represented in uppercase string format (i.e., `'A'`, `'C'`, `'G'`, or `'T'`).

   Function *extract*() has three parameters *ns*, *a*, and *b*, where *ns* is a string of nucleotides, and *a* and *b* are indices. The function returns a new string of nucleotides that is equal to the nucleotides of *ns* from indices *a* through *b*–1. Assume that $a \le b$.

   Function *snip*() has three parameters *ns*, *a*, and *b*, where *ns* is a string of nucleotides, and *a* and *b* are indices. The function returns a new string of nucleotides that is equal to *ns* without its nucleotides at indices *a* through *b*–1 inclusively. Assume that $a \le b$.

Function *complement* () has a parameter *ns*, where *ns* is a string of nucleotides. The function returns a string of nucleotides, each of whose nucleotides is the complement of the corresponding *ns* nucleotide.

The built-in tester makes use of the following nucleotide strings.

```
S1 = 'ACAGTCT'
S2 = 'ACCAACCCCGG'
S3 = 'TGAGTCCGAGGAGA'
S4 = 'GGGTGCTTCAGAG
```

And should produce the following output.

```
Test extract () yes or no? yes

extract( S1, 0, 3 ): ACA
extract( S2, 1, 4 ): CCA
extract( S3, 5, 9 ): CCGA
extract( S4, 2, 8 ): GTGCTT

Test snip() yes or no? yes


snip( S1, 0, 3 ): GTCT
snip( S2, 1, 4 ): AACCCCGG
snip( S3, 5, 9 ): TGAGTGGAGA
snip( S4, 2, 8 ): GGCAGAG

Test complement() yes or no? yes

complement( S1 ): TGTCAGA
complement( S2 ): TGGTTGGGGCC
complement( S3 ): ACTCAGGCTCCTCT
complement( S4 ): CCCACGAAGTCTC
```

7. Implement module *mm.py*. The module defines a function *two*(). The function has a dataset parameter *d*.

   The function returns a two-element list. The first element is the minimum value in the data set; the second element is the maximum value in the data set.

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

```
d1 = [ [ 2, 1 ] ]
d2 = [ [ "s", "i", "n", "g" ], [ "v", "i", "e" ], [ "d", "o" ] ]
d3 = [ [ 3, 1, 4 ], [1, 5, 9, 2], [ 6 ], [ 8 ], [ 10 ], [ -10 ] ]
```

   And should produce the following output.

```
two( d1 ): [1, 2]
two( d2 ): ['d', 'v']
two( d3 ): [–10, 10]
```

8. Implement module *double_vision.py*. The module defines a function *locate*(). Function *locate*() has two parameters *spot* and *original*, where *spot* is a coordinate and *original* is an `Image`.

   If *spot* lies on the lefthand side of *original*, then the function returns *spot*. Otherwise, the function returns the reflection location of *spot* with respect to the center vertical axis of *original*.

   The built-in tester should produce the following images.

9. Implement module *tweak.py*. The module defines a function *rgb*() with a pixel parameter *p*.

   The function returns a new pixel. The color of the new pixel depends upon the (*r*, *g*, *b*) values of *p*.

   - The result is (*r*, 0, 0), if *r* is equal to the maximum of *r*, *g*, and *b*.
   - If instead *g* is equal to the maximum of *g* and *b*, the result is (0, *g*, 0).
   - Otherwise the result is (0, 0, *b*).

The built-in tester should produce the following images.

10. Implement module *smallest*(). The module defines a function *second*(). The function has a list parameter *x*. The elements of *x* will be either be all strings or all numbers. Important: the function *does not* change *x*. So, if you need to make modifications, do it on a copy.

    The function returns the second smallest value in list *x*. Note, if the smallest value occurs more than once in *x*, then the second smallest value is the same as the smallest value.

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

```
x1 = [ 31, 4, 15, 9, 25 ]
x2 = [ 10, 20, 10 ]
x3 = [ "love", "is", "all", "you", "need" ]
x4 = [ "i", "am", "the", "walrus", "i", "am", "the", "walrus" ]
x5 = [ -31.25, -62.5, -125.0, -250.0, -500.0, -1000.0 ]
```

    And produces the following output.

```
second( x1 ): 9
second( x2 ): 10
second( x3 ): is
second( x4 ): am
second( x5 ): -500.0
```