Name:

Email id:

Pledge:

Notices

- Based on your past educational achievements, I expect you to do well on this test.
- Answer the questions in any order that you want.

Test rules

- Before you leave the room, check that you uploaded all 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 test is your laptop.
- Uploading after you leave the room means failing the class.
- Any cheating can result in failing the class and the incident being referred to the Honor Committee.
- Do not access class examples artifacts, web solutions, 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.

Modules

- Modules should follow class programming practices; e.g., whitespace, identifier naming, and commenting if you think it is needed, etc.
- Whether a program or function is runnable is important.
- None of your code should produce output. Comment out or delete all debugging `print()` statements before submitting.

1. Develop program *yes.py*

The program prints the word yes in all uppercase letters. The program does not print anything else.

2. Develop program *flakes.py*

The program prompts a user for the number of inches of snow that fell on Scott Stadium football field on Sunday. The program computes an estimate of the snowflakes that fell. The output of the program should be the numeric estimate and nothing else. For your information:

- The football field is 160 feet wide and 360 feet long.
- There are on average about 175 snowflakes per cubic inch of snow
- There are 12 inches per foot.

Below are two possible program runs.

```
Snow fall in inches: 8
11612160000
```

```
Snow fall in inches: 5
7257600000
```

3. Develop module *rat.py*

The module defines a function `ing()`, which has a single integer parameter `s`. Parameter s represents a wind speed in kilometers per hour (kph). The function returns the storm force category for such a wind speed according to an abbreviated version of the Beaufort scale:

- O kph is Calm
- 1 – 49 kph is a Breeze
- 50 – 117 kph is a Storm
- 118 or greater is a Hurricane

The output of the built-in tester should be:

```
ing( 1 ): Breeze
ing( 0 ): Calm
ing( 121 ): Hurricane
ing( 117 ): Storm
```

4. Develop module *grin.py*.

The module defines two functions `ana()` and `erse()`. Function `ana()` has two string parameters `s` and `t`; and function `erse()` has one string parameter `s`.

- Function `ana()` returns `True`, if its parameters `s` and `t` are anagrams of each other; otherwise, the function returns `False`. A string `t` is an *anagram* of string `s`, if it is formed by a rearrangement of the letters of `s`.

- Function `erse()` returns `True`, if word `s` and the word formed by reversing the letter order of `s` equal each other; otherwise, the function returns `False`.

The output of the built-in tester should be:

```
ana( live, vile ): True
ana( mountaineer , enumeration ): True
ana( feed, fed ): False
ana( done, need ): False

erse( deed ): True
erse( racecar ): True
erse( feed ): False
erse( palindrome ): False
```

5. Develop module *disj.py*

The module defines one function `oint()`, which has a single list parameter `a`. Parameter `a` is a list of strings without duplicates. The function returns `True` if there are pair of elements in `a`, where one of the elements is a substring of the other. Otherwise, the function returns `False`. The module has a built-in test tester. The tester output should be:

```
oint( ['a', 'b', 'cc', 'd'] ): False
oint( ['a', 'b', 'ca', 'd'] ): True
oint( ['a', 'b', 'bb', 'd'] ): True
oint( ['a', 'b', 'cd', 'd'] ): True
```

The easiest way to consider every pair of elements is through a nested loop.

6. Develop module *pals.py*

The module defines one function `f()`, which has a single integer parameter `n`. The function returns a list of the positive proper integer factors of `n`. The *factors* of a number are those integers that when dividing `n` have a remainder of 0. A factor is a *proper factor* if it is neither 1 nor `n`. The output of the built-in tester should be:

```
tors( 2 ): []
tors( 4 ): [2]
tors( 10 ): [2, 5]
tors( 48 ): [2, 3, 4, 6, 8, 12, 16, 24]
```

7. Develop module *george.py*

The module defines three functions `nand()`, `nor()`, and `xor()`. All three functions have two logical parameters `p` and q. And all three functions return a logical value of either `True` or `False`.

- Function `nand()` returns the complement of an *and* operation with parameters `p` and `q`; that is, if an `and` operation were to return `True`, function `nand()` returns `False`, and vice-versa.
- Function `nor()` returns the complement of an *or* operation with parameters `p` and `q`; that is, if an `or` operation were to return `True`, function `nor()` returns `False`, and vice-versa.

- Function `xor()` returns whether exactly one of its parameters has the value `True`.

The output of the built-in tester should be:

```
nand( False , False ): True
nand( False , True ): True
nand( True , False ): True
nand( True , True ): False

nor( False , False ): True
nor( False , True ): False
nor( True , False ): False
nor( True , True ): False

xor( False , False ): False
xor( False , True ): True
xor( True , False ): True
xor( True , True ): False
```

8. Develop module *dice.py*

The module defines two functions `roll()` and `rolls()` neither of which has any parameters. Function `roll()` returns a list of two integers and `rolls()` returns a list of eleven integers.

- Function `roll()` simulates the rolling of two six-sided dice. However, unlike normal dice, whose sides are numbered 1 through 6, the simulated dice are numbered 0 through 5.
- Function `rolls()` invokes function `roll()` thirty-six times. The function returns an integer list of size 11, where the i^{th} element of the list is the number of the rolls that added to i. My implementation initialized its accumulator to be an 11-element list.

The output of the built-in tester should be:

```
roll(): [2, 0]
roll(): [2, 4]
roll(): [0, 3]

rolls(): [1, 2, 2, 5, 3, 10, 6, 3, 2, 1, 1]
rolls(): [1, 2, 0, 5, 3, 4, 6, 5, 7, 1, 2]
rolls(): [1, 3, 2, 4, 9, 7, 1, 5, 1, 2, 1]
```

9. Develop module *fib.py*.

The module defines one functions `ona()`, which has a single integer parameter `n`. You can assume `n` is at least 2. The function returns an `n`-element list. The elements of the list are the first `n` Fibonacci numbers. By definition, the first two Fibonacci numbers are both 1. The successor Fibonacci numbers are always the sums of the two previous Fibonacci numbers.

The built-in tester output should be:

```
ona( 2 ): [1, 1]
ona( 5 ): [1, 1, 2, 3, 5]
ona( 6 ): [1, 1, 2, 3, 5, 8]
ona( 7 ): [1, 1, 2, 3, 5, 8, 13]
```

10. Develop module *reflect.py*.

The module defines a functions `loc()`. Function `loc()` takes two parameters `img` and `spot`, where `img` is an `Image` and `spot` is a coordinate. If `spot` lies on the lefthand side of `img`, then the function returns `spot`. Otherwise, the function returns the reflection location of `spot` with respect to the center axis of `img`.

The module has a built-in tester for the function that makes use of the image manipulation pattern. The image manipulation examines the below left image and if `loc()` is correct displays the below image on the right.

11. Develop module *age.py*.

The module defines one function `aver()`. Function `aver()` has one `Image` parameter `img`. The function returns a triple (`ar`, `ag`, `ab`), where `ar`, `ag`, and `ab` are respectively the integer averages of the red, green, and blue components of the RGB values for the pixels in `img`.

The function has a built-in tester that examines Thomas Jefferson and Mona Lisa images. The tester output should be:

```
age( Thomas Jefferson ): (96, 73, 45)
age( Mona Lisa ): (75, 61, 42)
```

12. Develop module *triad.py*.

The module defines functions `unique()`, `counts()`, and `cmp()`.

- Function `unique()` has a single integer list parameter `d`, and returns `True` if the values in list `d` are all different; otherwise , the function returns `False`. The function does not change list `d`.

- Function `counts()` has a single integer list parameter `d`. The function returns a new list, where the i^{th} element of the new list equals the number of occurrences of the i^{th} element of value in `d`. The function does not change list `d`.
- Function `cmp()` has two integer list parameters `d` and `e`. The function returns either -1, 0, or 1 depending respectively whether the sum of the elements of `d` is less than, equal to, or greater than the sum of the elements in `e`. The function changes neither list `d` nor `e`.

The output of the built-in tester should be:

```
unique( [31, 28, 31, 30] ): False
unique( [1, 24, 7, 52, 365] ): True

counts( [31, 28, 31, 30, 24] ): [2, 1, 2, 1, 1]
counts( [28, 30, 31, 30, 31, 31, 30, 31] ): [1, 3, 4, 3, 4, 4, 3, 4]

cmp( [1, 3, 5, 7] [17] ): -1
cmp( [1, 3, 5, 7] [4, 12] ): 0
cmp( [1, 3, 5, 7] [4, 5, 6] ): 1
```

13. Develop module *quad.py*.

The module defines a functions `sum()`, which has a dataset parameter `d`. The rows of dataset `d` all have four values. The function returns a 4-element list. The elements of the return list are respectively the sum of the first, second, third, and fourth columns `d`.

The built-in tester defines datasets `d1` and `d2`, where

`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, 987 ]]`

The output of the built-in tester should be:

```
sum( d1 ): [92, 89, 92, 92]
sum( d2 ): [273, 441, 714, 1155]
```