1. Create an assert statement that throws an AssertionError if the variable spam is a negative Integer.
Ans: assert spam >=0, 'Variable Spam should not be a -ve number'
2. Write an assert statement that triggers an AssertionError if the variables eggs and bacon contain strings that are the same as each other, even if their cases are different (that is, 'hello' and 'hello' are considered the same, and 'goodbye' and 'GOODbye' are also considered the same).
Ans:

```
In [10]:
def check(egg,bacon):
    egg = egg.upper()
    bacon = bacon.upper()
    assert not(egg == bacon), 'Eggs/Bacon should not be same'
```

```
In [11]:
check('hello','Hello')
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
C:\Users\THEJAS~1\AppData\Local\Temp/ipykernel_12160/4053998493.py in <module>
----> 1 check('hello','Hello')

C:\Users\THEJAS~1\AppData\Local\Temp/ipykernel_12160/375859929.py in check(egg, bacon)
      2     egg = egg.upper()
      3     bacon = bacon.upper()
----> 4     assert not(egg == bacon), 'Eggs/Bacon should not be same'

AssertionError: Eggs/Bacon should not be same
```

```
In [12]:
check('goodbye','GOODbye')
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
C:\Users\THEJAS~1\AppData\Local\Temp/ipykernel_12160/4167179568.py in <module>
----> 1 check('goodbye','GOODbye')

C:\Users\THEJAS~1\AppData\Local\Temp/ipykernel_12160/375859929.py in check(egg, bacon)
      2     egg = egg.upper()
      3     bacon = bacon.upper()
----> 4     assert not(egg == bacon), 'Eggs/Bacon should not be same'

AssertionError: Eggs/Bacon should not be same
```

```
In [8]:
egg = ''
bacon = ''
print('enter a string')
egg = input()
egg = egg.upper()
print('enter another string:')
bacon = input()
bacon = bacon.upper()
assert not(egg == bacon), 'Eggs/Bacon should not be same'
```

```
enter a string
red
enter another string:
RED

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
C:\Users\THEJAS~1\AppData\Local\Temp/ipykernel_12160/1798663917.py in <module>
      7 bacon = input()
      8 bacon = bacon.upper()
----> 9 assert not(egg == bacon), 'Eggs/Bacon should not be same'

AssertionError: Eggs/Bacon should not be same
```

3. Create an assert statement that throws an AssertionError every time.
Ans:

```
In [13]:
def assert_always():
    assert False, 'Assertion Error'
assert_always()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
C:\Users\THEJAS~1\AppData\Local\Temp/ipykernel_12160/3829148055.py in <module>
      1 def assert_always():
      2     assert False, 'Assertion Error'
----> 3 assert_always()

C:\Users\THEJAS~1\AppData\Local\Temp/ipykernel_12160/3829148055.py in assert_always()
      1 def assert_always():
----> 2     assert False, 'Assertion Error'
      3 assert_always()

AssertionError: Assertion Error
```

4. What are the two lines that must be present in your software in order to call logging.debug()?

Ans:

```
In [14]:
import logging
logging.basicConfig(filename = 'application_log.txt',level=logging.DEBUG,
                    format=' %(asctime)s - %(levelname)s - %(message)s')
```

5. What are the two lines that your program must have in order to have logging.debug() send a logging message to a file named programLog.txt?

Ans:

```
In [ ]:
import logging
logging.basicConfig(
    filename='programLog.txt',
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

```

6. What are the five levels of logging?

Ans: Debug, info, warning, error, critical.

7. What line of code would you add to your software to disable all logging messages?

Ans: logging.disable = True

8.Why is using logging messages better than using print() to display the same message?

Ans: Logging allows you to categorize messages and turn them on or off depending on what you need without removing the logging function. Print statements are not easily managed.

9. What are the differences between the Step Over, Step In, and Step Out buttons in the debugger?

Ans: Step in causes the debugger to execute the next line of code and then pause again.
Step Over executes the next line of code, if the next line of code is a function call, the Step Over button will “step over” the code in the function. The debugger will pause as soon as the function call returns.
Step out causes the debugger to execute lines of code at full speed until it returns from the current function.

10.After you click Continue, when will the debugger stop ?

Ans: Until it terminates or reaches a breakpoint in the code.

11. What is the concept of a breakpoint?

Ans: Breakpoint is a setting on a line of code that causes the debugger to pause when the program execution reaches the line.