1. Create an assert statement that throws an AssertionError if the variable spam is a negative integer.

In [3]:
```
# AssertionError with error_message.
x = 1
y = 0
assert y != 0, "Invalid Operation" # denominator can't be 0
print(x / y)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In [3], line 4
      2 x = 1
      3 y = 0
----> 4 assert y != 0, "Invalid Operation" # denominator can't be 0
      5 print(x / y)

AssertionError: Invalid Operation
```

The assert statement consists of the assert keyword followed by a condition that is expected to be true.

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).

In [4]:
```
assert eggs.lower() != bacon.lower(), "eggs and bacon should have different string val
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [4], line 1
----> 1 assert eggs.lower() != bacon.lower(), "eggs and bacon should have different s
tring values"

NameError: name 'eggs' is not defined
```

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

In [5]:
```
assert False, "This assert statement always raises an AssertionError"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In [5], line 1
----> 1 assert False, "This assert statement always raises an AssertionError"

AssertionError: This assert statement always raises an AssertionError
```

The assert statement checks if a condition is true, and if it's not true, it raises an AssertionError with an optional error message. In this case, we're passing False as the condition, which is always false, so the assert statement will always raise an AssertionError with the message "This assert statement always raises an AssertionError".

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

In [6]:
```
#In order to call the logging.debug() method, we must include the following two lines
import logging
logging.basicConfig(level=logging.DEBUG)
```

The first line imports the logging module, which provides a flexible logging system in Python. The second line configures the root logger to log messages at the DEBUG level or higher. This means that any messages logged at the DEBUG level or higher (i.e., messages with severity levels of DEBUG, INFO, WARNING, ERROR, or CRITICAL) will be output to the console.

Once you have these two lines in your script, you can call logging.debug() to log a message at the DEBUG level.

In [7]:
```
import logging
logging.basicConfig(level=logging.DEBUG)

# ...

logging.debug("This is a debug message")
```

```
DEBUG:root:This is a debug message
```

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?

In [8]:
```
#To have logging.debug() send a logging message to a file named programLog.txt, we wou

import logging

logging.basicConfig(filename='programLog.txt', level=logging.DEBUG)
```

The first line imports the logging module. The second line configures the root logger to log messages at the DEBUG level or higher, and also sets the output to be written to a file named programLog.txt instead of the console.

Once we have these two lines in your script, you can call logging.debug() to log a message at the DEBUG level, and it will be written to the programLog.txt file

In [9]:
```
import logging

logging.basicConfig(filename='programLog.txt', level=logging.DEBUG)

# ...

logging.debug("This is a debug message")
```

```
DEBUG:root:This is a debug message
```

6. What are the five levels of logging?

There are five standard levels of logging that are available in the Python logging module. They are, in increasing order of severity:

DEBUG: Detailed information, typically of interest only when diagnosing problems.

INFO: Confirmation that things are working as expected.

WARNING: An indication that something unexpected or potentially problematic has occurred.

ERROR: An indication that an error or exception has occurred, but the program can recover from it.

CRITICAL: A very serious error, indicating that the program may not be able to continue running.

By default, the logging module will capture all messages at the WARNING level or higher. However, you can configure the logging level to capture messages at a specific level or higher by setting the logging level to that level or higher. For example, logging.basicConfig(level=logging.DEBUG) would capture all messages at the DEBUG level or higher.

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

In [10]:
```
#To disable all logging messages in your software, you can add the following line of c

logging.disable(logging.CRITICAL)
```

This line of code sets the logging level to CRITICAL, which is the highest severity level.

Since all logging messages with a severity level lower than CRITICAL will be ignored, this effectively disables all logging messages.

we will need to have import logging at the beginning of your code for this line to work.

It's important to note that this line of code should be used with caution, as it disables all logging messages, including critical error messages that may be important for debugging and troubleshooting your code.

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

Using logging messages is generally better than using print() to display the same message because:

Logging messages are more flexible: The logging module provides more flexibility than print() when it comes to formatting and handling log messages. You can easily customize the output format, set the logging level for each message, and even write messages to different destinations (e.g., console, file, database).

Logging messages can be turned on/off: With logging, you can easily turn off all logging messages or selectively disable messages at a specific logging level. This can be very helpful when you need to temporarily disable logging messages in production, or when you want to focus on a specific logging level during development.

Logging messages are more professional: Logging messages provide a more professional and organized way of reporting errors and status messages than just using print() statements. They make it easier for developers to understand what's happening in the code, and to troubleshoot errors and issues.

Logging messages are more performant: When you use print() statements to output messages, the messages are written to the console immediately. This can slow down your program if you're logging a lot of messages. With logging, messages are buffered and can be written to the console or file in batches, which can improve the performance of your program.

Overall, using logging messages is a better practice than using print() statements when you want to report errors, status messages, or any other information from your code.

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

The Step Over, Step In, and Step Out buttons in a debugger are used to control the execution of code line by line during debugging. Here are the differences between these buttons:

Step Over: This button executes the current line of code and moves to the next line of code. If the current line of code contains a function call, the function call is executed, but the debugger doesn't step into the function. This button is useful when you want to skip over a function call and continue debugging the code outside of the function.

Step In: This button executes the current line of code and moves to the next line of code, just like Step Over. However, if the current line of code contains a function call, the debugger steps into the function and starts debugging the code inside the function. This button is useful when you want to debug a function in detail and step through each line of code inside the function.

Step Out: This button is used to execute the remaining lines of code in the current function and return to the calling function. If you're inside a function and you want to quickly return to the calling function without stepping through each line of code in the function, you can use the Step Out button.

In summary, the Step Over button skips over function calls, the Step In button steps into function calls, and the Step Out button returns from a function call to the calling function. These buttons are useful for controlling the execution of code line by line during debugging and can help you identify and fix issues in your code.

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

After we click Continue in a debugger, the debugger will stop when it encounters a breakpoint or an exception is raised. Here's what happens in each case:

Breakpoint: A breakpoint is a specific point in your code where you want the debugger to stop so that you can examine the state of the program. When you set a breakpoint, the debugger will stop execution at that point and allow you to step through the code line by line. If you click Continue after hitting a breakpoint, the debugger will continue executing the code until it encounters the next breakpoint or an exception is raised.

Exception: An exception is an error that occurs during the execution of your code. When an exception is raised, the debugger will stop execution at the line of code where the exception occurred and allow you to examine the state of the program. If you click Continue after hitting an exception, the debugger will continue executing the code until it encounters the next exception or a breakpoint.

It's important to note that if there are no breakpoints or exceptions in your code, the debugger will continue executing the code until it reaches the end of the program. In this case, the debugger will not stop unless you explicitly stop it or the program terminates.

11. What is the concept of a breakpoint?

A breakpoint is a debugging tool that allows us to pause the execution of your code at a specific line or point in your code. When you set a breakpoint, the debugger will stop executing your code at that point and allow you to examine the state of your program, including the values of variables, the call stack, and other information that can help you identify and fix issues in your code.

Setting a breakpoint is typically done through an integrated development environment (IDE) or a debugger interface, where you can click on the line of code where you want the breakpoint to be set. Once a breakpoint is set, you can run your code in debug mode, and the debugger will stop execution at the breakpoint. You can then step through your code line by line, examine the state of your program, and make changes as needed.

Breakpoints are useful when we're debugging complex code, as they allow you to isolate specific parts of your code and focus on them individually. They can also help us identify issues that may only occur under specific conditions or during specific parts of your program's execution.

Overall, breakpoints are a powerful debugging tool that can help us identify and fix issues in our code more quickly and efficiently.