**CS1101 – Programming Fundamentals Assignment 3**

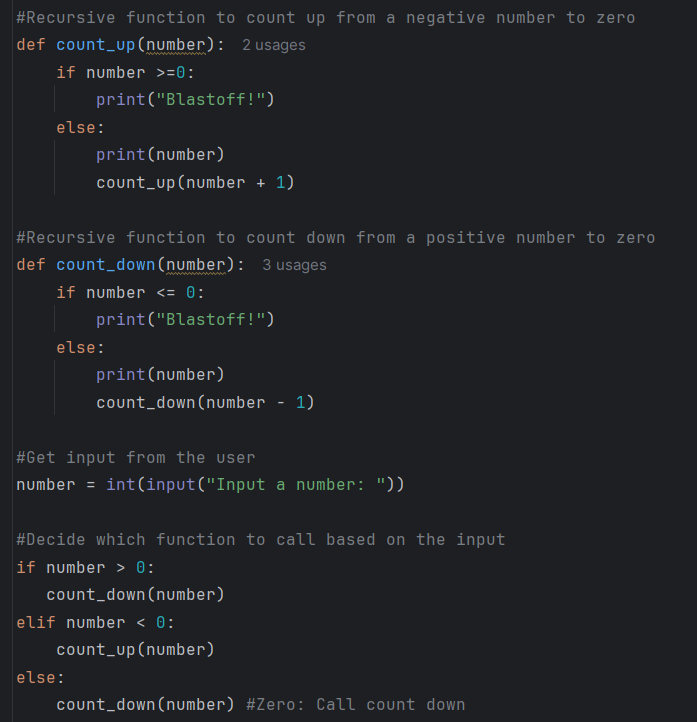
**Student Name:** Merhawit Kahsay Gidey  
**Course:** CS1101 – Programming Fundamentals  
**Instructor:** Henry hu  
**Date:** Jul 7, 2025

**Question 1**

This Python program uses recursion to either count **up** from a negative number or **down** from a positive number until it reaches zero, at which point it prints "Blastoff!". It defines two recursive functions: countup(n) and countdown(n). The countup function is called when the user inputs a negative number and works by printing the current number, then calling itself with the number increased by one, moving closer to zero.

The countdown function is called when the input is positive and prints the number, then calls itself with the number decreased by one. If the user enters zero, the program defaults to calling the countdown function, which immediately prints "Blastoff!". This is based on the reasoning that countdowns commonly start or end at zero, such as in rocket launches.

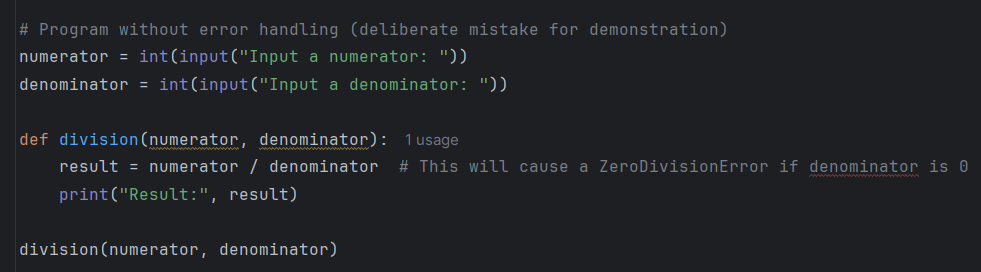
The program uses input() to get the number from the user, checks whether it is positive, negative, or zero, and then calls the appropriate function accordingly.



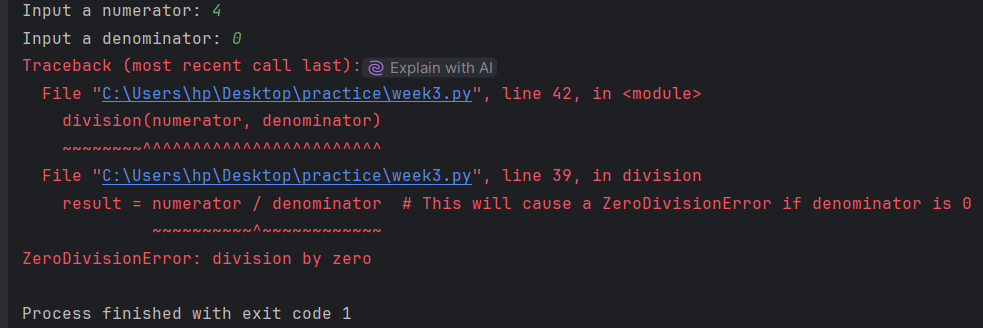
When the user enters 0, I chose to call the countdown(0) function because it aligns better with the natural way we understand countdowns in real life. In the countdown function, if the input is zero or less, it immediately prints "Blastoff!", which reflects what typically happens at the end of a countdown like when launching a rocket or starting a race. Zero is traditionally seen as the **end point** in a countdown sequence (e.g., 3...2...1...0...Blastoff!).

**Question 2: *Division by Zero Error Handling in Python***

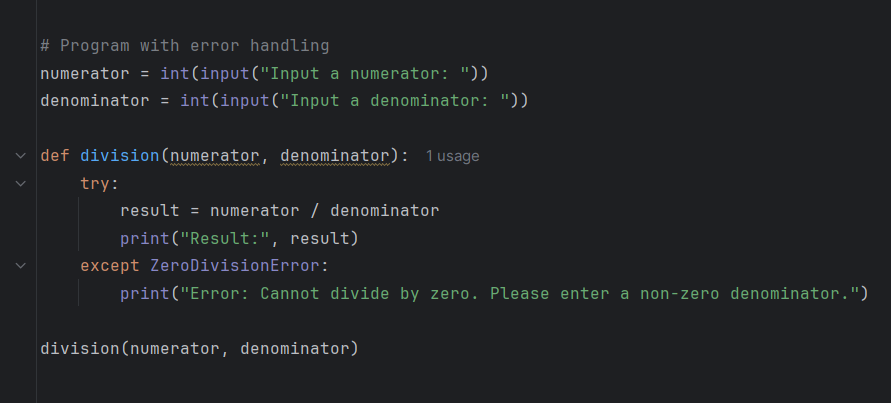
In programming, runtime errors are common occurrences that interrupt the execution of a program, especially when operations involve user input. One of the most frequent runtime errors in Python is the **division by zero error**, which occurs when a program attempts to divide a number by zero, a mathematically invalid operation. This guide is designed to help junior developers understand how such errors happen, how to identify them, and most importantly, how to handle them gracefully using Python’s error-handling techniques. By deliberately creating and then fixing a program that divides two user-provided numbers, we demonstrate how to implement robust and user-friendly solutions to prevent program crashes and improve code reliability.

The original version of the program prompts the user to enter two numbers and then attempts to perform a division operation without checking if the denominator is zero. This results in a runtime error if the user inputs zero for the denominator. Here's the unhandled version of the code:

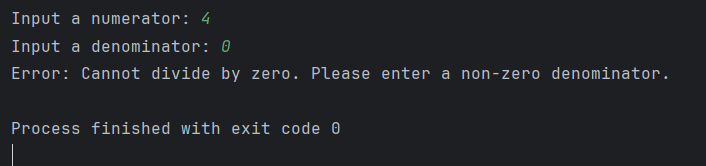
When the user inputs 0 as the denominator, the program throws a ZeroDivisionError, as shown below:

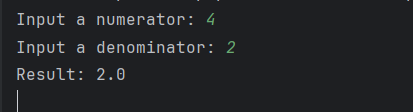


To handle this issue, we use a try-except block to catch the error and respond with a friendly message, preventing the program from crashing. Here is the corrected version:



If the user now inputs 0 as the denominator, the output is:





**Significance of Error Handling in Expressions or Conditions**

Error handling is crucial in programming because it allows a program to recover from unexpected input or behavior without crashing. In this example, the division by zero error is a classic case where an unchecked operation could terminate the entire program, making it appear unreliable or incomplete. By using try and except statements, we can catch such errors, provide clear feedback to the user, and guide them to input correct values, all without stopping the program’s flow. This improves the user experience and prevents potential data loss, confusion, or misbehavior of the software. In larger systems, failing to handle division by zero could even affect calculations in financial, medical, or engineering applications, leading to serious consequences.

**Teaching Junior Developers to Handle Errors**

Junior developers must be encouraged to expect errors as a normal part of programming and to build defenses around common risk points - like division, file access, or type conversions. By showing how unhandled exceptions disrupt the program and contrasting that with handled exceptions that guide the user, they begin to understand the power and responsibility of writing robust, defensive code. Clear, user-friendly error messages and control over the program’s behavior are hallmarks of professional software, and mastering try-except structures is one of the first steps toward writing reliable Python programs.

Reference

Downey, A. (2015). Think Python: How to Think Like a Computer Scientist. [Green Tea Press](https://greenteapress.com/thinkpython2/thinkpython2.pdf). <https://greenteapress.com/thinkpython2/thinkpython2.pdf>