**Conditionals, Recursion, and Error Handling**

Kenneth Thomas Barrett

Computer Science Department, University of the People

CS1101 - Programming Fundamentals

Ola Fatai Ayanleke Ayanleke

December 1, 2024

### Q1

The GitHub link to this question’s code can be found [here](https://github.com/KennethTBarrett/CS1101/blob/main/Unit_3/unit_3_assignment_Q1.py). While the code comments explain my choice of which function to call for an input of zero, I’ve also provided explanations below.

#### Explanation of the Code

To start, I used the classic conditional “if \_\_name\_\_ == "\_\_main\_\_":” as an entry point to call the primary function for this script. This indicates that the script is being directly executed rather than imported as a module in another script and means that the primary function will automatically trigger upon the script's execution. I use a while loop to continuously prompt the user to input a value until they provide a valid input (in this case, an integer). Once a valid input has been received, the break statement exits the loop, and the program proceeds. Additionally, I use a try block to attempt to execute code that may raise an exception and handle those exceptions gracefully. Within this block, the user is prompted to enter an integer. If the user inputs an invalid entry (in this case, a value that cannot be converted to an integer, such as “abc”), a ValueError exception is raised, telling the user to enter a whole number.

Assuming the user has entered a valid input, I use conditional statements to determine if the value is more than zero, less than zero, or precisely zero. I call the countdown() function if the value is greater than zero. This function checks if the value is 0 and returns ‘Blastoff!’ if so. If not, the function prints the current value and makes a recursive call to itself, decreasing the value by 1. Due to the nature of recursion, this will continue until the value equals zero. On the other hand, if the value is less than zero, the countup() function is called. This function follows the same logic but increases the value with each recursive call instead of decreasing it.

#### Explanation for My Choice for an Input of Zero

I opted to use the countdown function based upon a coin-flip; because both the countup and countdown functions have a condition to return 'Blastoff' when n = 0, it doesn't make a difference which function is used. However, given complete control of the code structure, I would simply print 'Blastoff!' in the main() function. This enhances code readability and provides a more complete logical structure of the function, reducing the chance of confusing other developers upon their first glance at the code.

### Q2

The GitHub link to this question’s code can be found [here](https://github.com/KennethTBarrett/CS1101/blob/main/Unit_3/unit_3_assignment_Q2.py). Like Q1, the answers to the questions relative to Q2 can be found in comments within my code, although I’ve provided explanations here as well.

#### Explanation of the Code

Like the code in Q1, the code for Q2 uses an entry point to call the primary function of the script and uses a while loop and a try block to accept input from the user, prompting them to enter a numerator and denominator. I accepted the input as a float, using float(input(...)). This allows the user to input either an integer or a floating-point number. Additionally, I created a function ‘format\_num()’ to accept the input and determine if it is a whole number or a floating-point number by using the modulo operator to examine if dividing the value by 1 results in a remainder. If it did not, I converted and returned the value as an integer. If it did, I returned the value as it was input. This way, a whole number can be printed without a decimal while retaining the user’s ability to input a floating-point number, keeping the output as visually clean as possible while expanding functionality.

I then used a conditional statement to check whether the denominator entered by the user was 0. A ValueError was raised if it was, informing the user that numbers cannot be divided by zero. The calculations were then performed within a print statement, utilizing the format\_num() function if the quotient resulted in a whole number.

#### Runtime Error for Dividing By Zero

If a user attempts to divide by zero, the following error will be thrown:

*‘ZeroDivisionError: division by zero’*

As a result, I created a condition that raises a ValueError informing the user that division by zero is not possible. It is crucial to notify the user why the error has been thrown and provide actionable information on how to fix it moving forward. By raising a ValueError if a user enters 0 as the value of the denominator, I can provide a print statement telling them that division by zero is not possible, giving them actionable insight into what is causing the error.

#### The Importance of Error Handling

Error handling helps ensure the software is robust and user-friendly, making it more prepared to handle unexpected or invalid conditions gracefully. Error handling can prevent crashes and provide feedback to users on how to guide the software to a valid result. In the case of a ZeroDivisionError, the program will crash if not handled, potentially leading to data loss and creating a frustrating user experience. When this error is appropriately handled, the user is informed about what went wrong and how to fix it. Additionally, a division by zero error could corrupt calculations in software dealing with financial data. It can expose sensitive information (such as stack traces) that may decrease data integrity and security. Error handling helps validate inputs before performing operations and can provide alternative actions to fall back on. Aside from the users, error handling helps provide a structured way to log issues. This can improve debugging capabilities and provide a way for developers to track recurring problems when the software is in production.

To summarize, error handling is vital for ensuring end users have a positive experience by preventing crashes, clearly informing users of the cause of errors, and guiding them on how to solve them. It also helps ensure the stability and security of the software and the data it collects and processes. Lastly, error handling helps facilitate debugging efforts and informs long-term software maintenance.

### Final Notes

#### Outputs

Sample outputs can be found at the bottom of each script.

#### References

No sources have been provided as this knowledge has been attained through my professional experience. In particular, emphasis on extensive error handling was crucial to my last development position in a tax software company where data security and integrity were critical.