HyperionDev

Defensive Programming – Error Handling

Model-Answer Approach

Visit our website

Auto-graded task 1

In both `errors.py` and `errors2.py`, the focus is on identifying and correcting compilation errors, runtime errors, and logical errors.

For `errors.py,` the approach begins with fixing compilation errors, such as missing parentheses around the `print()` statement and correcting unexpected indentation. Python enforces strict indentation rules, and any deviation from these rules leads to syntax errors, which were resolved by ensuring the correct alignment of the code blocks. Furthermore, a runtime error occurred due to improper use of the equality operator `(==)` instead of the assignment operator `(=)`, which was fixed by assigning the string `"24 years old"` to the variable. Another runtime error was caused by attempting to convert a string containing non-numeric characters into an integer, which was fixed by slicing the string to remove non-numeric parts before conversion. Finally, a logical error occurred when trying to concatenate a string with an integer without casting the integer to a string, which was fixed by using `str()` around the integer.

For `errors2.py`, similar errors were encountered and resolved. A compilation error occurred due to missing quotes around the string `"Lion"`, which was fixed by enclosing the string properly. Additionally, a logical error in the string formatting was identified, where the placeholders in the format string did not match the corresponding variables. Using Python's f-string formatting, this issue was corrected by ensuring that the variables `{animal}`, `{animal_type}`, and `{number_of_teeth}` were correctly inserted in the string to generate the desired output. As with the previous file, compilation errors related to missing parentheses around the `print()` statement were addressed to ensure proper execution in Python 3.

Both programs highlight common issues faced in early programming, such as syntax, type conversion, and string formatting errors. The approach involved systematically identifying each error type, providing a clear explanation for why the error occurred, and applying the appropriate fix. One potential pitfall is ensuring that, beyond fixing syntax and runtime errors, the logic behind the program is sound. Logical errors, in particular, can be harder to detect as they do not stop the program from running but produce incorrect results, so attention to detail in variable usage and formatting is essential.

Auto-graded task 2

This program prompts the user to enter their age and output a relevant message based on their age group. It checks the user's age and prints a message about whether they are a minor, a toddler, or a major. However, there is a logical flaw in the ordering of the conditions. The first condition checks if the age is under 18, which would be true for any age below 18, including those that should fall into more specific categories, such as

toddlers or minors under 10. For instance, if a user inputs the age "1," the program would output "You are a minor," even though it should more accurately classify them as a toddler.

This issue arises because the broader condition `(age < 18)` is placed before more specific conditions `(age < 10, age < 5, age < 3)`. Once the first condition is true, the remaining conditions are not checked, resulting in incorrect output for younger ages. This can be resolved by reordering the conditions from the most specific to the least specific, ensuring that the program checks for toddler or minor subcategories before considering the general "minor" category.

A possible pitfall when working with multiple conditions like this is the risk of misordering or overlapping conditions, where broader conditions are checked before more specific ones, leading to unintended behaviour. In this case, the incorrect ordering causes younger ages to be misclassified. Additionally, ensuring accurate condition logic is crucial for cases involving overlapping ranges, which can easily introduce logical errors into conditional structures.

Optional challenge

The approach for this task focuses on demonstrating the identification and resolution of different types of errors: compilation, runtime, and logical errors. The program begins with two compilation errors, which occur when the Python interpreter cannot process the syntax correctly. The first compilation error is a syntax error caused by missing parentheses in the `print` statement, which are mandatory in Python 3. The second compilation error involves a `NameError` due to the use of an undefined variable, `number`, highlighting the importance of declaring variables before using them in the code.

Next, the task introduces a runtime error, which occurs when the program encounters an issue during its execution despite being free of syntax errors. In this case, the runtime error is a `ZeroDivisionError`, caused by an attempt to divide by zero, an operation that is mathematically undefined. This demonstrates how a program can crash during execution if runtime errors are not handled properly.

Finally, the logical error is illustrated through flawed program logic that classifies 18-year-olds as adults instead of minors. The condition used `(age < 18)` is incorrect as it fails to include 18-year-olds as minors. To correct this logical error, the condition should be changed to `age <= 18` to accurately reflect the classification intent. This demonstrates how logical errors result in incorrect program behaviour, even though the code runs without generating any runtime or compilation errors.

A potential pitfall is not understanding the distinction between runtime and logical errors, as both can allow the program to execute initially. While runtime errors halt the program during execution, logical errors allow it to run but produce incorrect results. Remember to carefully analyse the program's logic to ensure it behaves as intended

and to consider implementing error-handling techniques (such as using `try/except` blocks) to prevent runtime errors from causing the program to crash.