TASK

Defensive Programming - Exception Handling

Visit our website

Introduction

WELCOME TO THE DEFENSIVE PROGRAMMING - EXCEPTION HANDLING TASK!

Debugging is essential to any Software Developer! In this task, you will learn about some of the most commonly encountered errors. You will also learn to use your IDE to debug your code more effectively.

WHAT IS DEFENSIVE PROGRAMMING?

As you may know, defensive programming is an approach to writing code where the programmer tries to anticipate problems that could affect the program and then takes steps to defend the program against these problems. MANY problems could cause a program to run unexpectedly!

Some of the more common types of problems to look out for are listed below:

- **User errors:** the people that use your application will act unexpectedly! They will do things like entering string values where you expect numbers. Or they may enter numbers that cause calculations in your code to crash (e.g. the prevalent divide by zero error). Otherwise, they may press buttons at the wrong time or more times than you expect. As a developer, anticipate these problems and write code that can handle these situations. For example, one approach is to check all user input.

- **Errors caused by the environment:** write code that will handle errors in the development and production environment. For example, in the production environment, your program may get data from a database that is on a different server. Code should be written in such a way that it deals with the fact that some servers may be down, the load of people accessing the program is higher than expected, etc.

- **Logical errors with the code:** Besides external problems that could affect a program, a program may also be affected by errors within the code. All code should be thoroughly tested and debugged.

IF STATEMENTS FOR VALIDATION

Many of the programming constructs that you have already learned can be used to write defensive code. For example, if you assume that any user of your system will be younger than 150 years old, you could use a simple `if` statement to check that someone enters a valid age.

EXCEPTION HANDLING

Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.

For instance, they occur when we try to open a file (for reading) that does not exist (`FileNotFoundError`), try to divide a number by zero (`ZeroDivisionError`) or try to import a module that does not exist (`ImportError`).

Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

The following examples illustrate exceptions occurring when trying to divide by zero, using a variable that has not been declared, and trying to concatenate a string with an integer:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The above exceptions are all built into the Python language and are just some of the many built-in exceptions. To find out more about the various built-in exceptions, you can take a look at the **documentation for Python 3**.

A part of defensive coding is to anticipate where these exceptions might occur, make provisions for such exceptions, and trigger specific actions for when these errors occur.

TRY - EXCEPT BLOCK

When handling exceptions, there are two main blocks of code that are used to ensure that specific actions are taken when the exception occurs, namely the *try block* and the *except block*. The try block is the part of the code that we are trying to execute if no exceptions occur. The except block, therefore, is the code that we specify needs to be executed if an exception does occur. Let's look at the following example:

```
while True:
    try:
        x = int(input("Please enter an integer: "))
        break
    except ValueError:
        print("Oops!  That was not a valid entry.  Try again...")
```

In the above code example, we have a while loop used to take user input. The try block gets executed first. If the input is a valid number, the integer value will be assigned to the "**x**" variable, and the loop will break.

However, if the user input is a string that is not a valid number, a `ValueError` is raised. The except block will then be executed, thereby printing the message to the terminal. It is worth noting that any code can be executed in the except block, and it does not have to be a print statement. After the message is printed to the terminal, the loop will start from the try block again until the user enters a valid input, and the loop can break.

We can also perform the above task in the following way:

```
while True:
    try:
        x = int(input("Please enter an integer: "))
        break
    except Exception:
        print("Oops!  That was not a valid entry.  Try again...")
```

This method is to be used cautiously as a fundamental programming error can slip through in your programs this way. The above code will run the *except block* no matter what type of exception occurs from the execution of the *try block*.

TRY - EXCEPT - FINALLY

There are occasions where a block of code needs to be executed whether or not an exception has occurred. In this scenario, we can use the *finally block*. The *finally block* is usually used to terminate anything after it has been utilised such as database connections or open resources. In the example below, the file object needs to be closed whether an exception has occurred or not; therefore, the file object is closed in the *finally block*.

```
file = None
try:
    file = open('input.txt', 'r')
    # Do stuff with the file here

except FileNotFoundError:
    print("The file that you are trying to open does not exist")

finally:
    if file is not None:
       file.close()
```

In the code example above, defensive programming is used. The coder has anticipated that the file may not be found. Therefore, the code handles the exception.

Beware! Do not be tempted to overuse *try-except* blocks! If you can anticipate and fix potential problems without using *try-except* blocks, do so. For example, don't use *try-except* blocks to avoid writing code that validates user input.

EXCEPTION OBJECTS

When an exception occurs, an exception object is created. The exception object contains information about the error. We can get more information about this error by printing the error object.

Let's use our previous example, but this time we are going to assign the exception object to a variable by using the *except-as* syntax and then print the error:

```
file = None
try:
    file = open('input.txt', 'r')
    # Do stuff with the file here

except FileNotFoundError as error:
    print("The file that you are trying to open does not exist")
    print(error)
finally:
    if file is not None:
       file.close()
```

When this code executes and the file does not exist, then the following output is generated which gives us some more information about the error that occurred:

```
The file that you are trying to open does not exist
[Errno 2] No such file or directory: 'input.txt'
```

RAISING EXCEPTIONS

There will be occasions when you want your program to raise a custom exception whenever a certain condition is met. In Python we can do this by using the "`raise`" keyword and adding a custom message to the exception:

In the example below, we are asking the user to input a value greater than 10. If the user enters a number that does not meet that condition, an exception is raised with a custom error message:

```
num = int(input("Please enter a value greater than 10: "))
if num <= 10:
    raise Exception(f'num value ({num}) is less than or equal to 10')
```

Instructions

Read the instructions provided in your compulsory task below. Take your time – you're putting a number of concepts together for this task.

Compulsory Task 1

Follow these steps:

- Create a simple calculator application called **calc_app.py** that performs and records simple calculations.
- The calculator application should allow users to perform a calculation or print previous calculations stored in a file called **equations.txt**.
  - If a user chooses to perform a calculation, the app should accept two numbers and an operation (**+**, **-**, *****, or **/**) as inputs from the user. The answer should be displayed for the user, and the equation and answer should be recorded in equations.txt (e.g. 21 + 3 = 24).
    - Use defensive programming to write a robust program that handles **unexpected** events and user inputs.
  - If a user chooses to print previous equations from equations.txt, display all previous equations.
    - Use defensive coding to ensure the program **does not crash** if the file does not exist.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.