HyperionDev

Defensive Programming – Exception Handling

Task

Visit our website

Introduction

Debugging is essential to any software developer. In this task, you will learn about some of the most commonly encountered errors by going through the concepts of defensive programming and exception handling. You will also learn to use your IDE to debug your code more effectively.

Defensive programming

Defensive programming is an approach to writing code in which 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 most common types of problems to look out for are listed below:

- **User errors:** The people who 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 `ZeroDivisionError`). They may also 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, by checking 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 to deal with the fact that some servers may be down, or 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.

Using `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 entered a valid age.

Handling exceptions

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 (`ModuleNotFoundError`).

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, you can take a look at the **documentation on built-in exceptions 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 they occur.

`try-except` block

When handling exceptions, two main blocks of code 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 try to execute if no exceptions occur. The `except` block, on the other hand, is the code that we specify needs to be executed if an exception does occur. Let's look at the following example:

```python
# Continuously prompt the user until a valid input is provided
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 used a `while` loop to take user input. The `try` block gets executed first, and 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 will break.

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

```python
# Continuously prompt the user until a valid input is provided
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 should be used cautiously as a fundamental programming error can slip through in 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.

```python
# Initialise the file variable to None
file = None

# Try block to attempt opening and processing the file
try:
    file = open('input.txt', 'r')
    # Do stuff with the file here

# Handle the case where the file does not exist
except FileNotFoundError:
    print("The file that you are trying to open does not exist")

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

Defensive programming was used in the code example above. The coder has anticipated that the file may not be found, therefore, the code handles the exception.

Code hack

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

Understanding exception objects

When an exception occurs, an exception object is created. The exception object contains information about the error, and 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 that 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')
```

Take note

The task below is **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, and you do not need to wait for it to be reviewed by a mentor. You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects. In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task

- 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.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.