Exception Handling

Estimated time needed: **15** minutes

Objectives

After completing this lab you will be able to:

- Understand exceptions
- Handle the exceptions

Table of Contents

What is an Exception?

In this section you will learn about what an exception is and see examples of them.

Definition

An exception is an error that occurs during the execution of code. This error causes the code to raise an exception and if not prepared to handle it will halt the execution of the code.

Examples

Run each piece of code and observe the exception raised

```
1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipykernel_74/2354412189.py in <module>
----> 1 1/0

ZeroDivisionError: division by zero
```

ZeroDivisionError occurs when you try to divide by zero.

```
y = a + 5
```

NameError -- in this case, it means that you tried to use the variable a when it was not defined.

```
a = [1, 2, 3]
a[10]
```

IndexError -- in this case, it occured because you tried to access data from a list using an index that does not exist for this list.

There are many more exceptions that are built into Python, here is a list of them https://docs.python.org/3/library/exceptions.html

Exception Handling

In this section you will learn how to handle exceptions. You will understand how to make your program perform specified tasks instead of halting code execution when an exception is encountered.

Try Except

A try except will allow you to execute code that might raise an exception and in the case of any exception or a specific one we can handle or catch the exception and execute specific code. This will allow us to continue the execution of our program even if there is an exception.

Python tries to execute the code in the try block. In this case if there is any exception raised by the code in the try block, it will be caught and the code block in the except block will be executed. After that, the code that comes after the try except will be executed.

```
# potential code before try catch

try:
    # code to try to execute
except:
    # code to execute if there is an exception

# code that will still execute if there is an exception
```

Try Except Example

In this example we are trying to divide a number given by the user, save the outcome in the variable a, and then we would like to print the result of the operation. When taking user input and dividing a number by it there are a couple of exceptions that can be raised. For example if we divide by zero. Try running the following block of code with b as a number. An exception will only be raised if b is zero.

```
a = 1

try:
    b = int(input("Please enter a number to divide a"))
    a = a/b
    print("Success a=",a)
except:
    print("There was an error")
```

Try Except Specific

A specific try except allows you to catch certain exceptions and also execute certain code depending on the exception. This is useful if you do not want to deal with some exceptions and the execution should halt. It can also help you find errors in your code that you might not be aware of. Furthermore, it can help you differentiate responses to different exceptions. In this case, the code after the try except might not run depending on the error.

Do not run, just to illustrate:

```
# potential code before try catch

try:
    # code to try to execute
except (ZeroDivisionError, NameError):
    # code to execute if there is an exception of the given types

# code that will execute if there is no exception or a one that we are
handling

# potential code before try catch

try:
    # code to try to execute
except ZeroDivisionError:
    # code to execute if there is a ZeroDivisionError
except NameError:
    # code to execute if there is a NameError

# code that will execute if there is no exception or a one that we are
handling
```

You can also have an empty except at the end to catch an unexpected exception:

Do not run, just to illustrate:

```
# potential code before try catch

try:
    # code to try to execute
except ZeroDivisionError:
    # code to execute if there is a ZeroDivisionError
except NameError:
    # code to execute if there is a NameError
except:
    # code to execute if ther is any exception
# code that will execute if there is no exception or a one that we are
handling
```

Try Except Specific Example

This is the same example as above, but now we will add differentiated messages depending on the exception, letting the user know what is wrong with the input.

```
a = 1

try:
    b = int(input("Please enter a number to divide a"))
    a = a/b
    print("Success a=",a)
except ZeroDivisionError:
    print("The number you provided cant divide 1 because it is 0")
except ValueError:
    print("You did not provide a number")
except:
    print("Something went wrong")
```

Try Except Else and Finally

else allows one to check if there was no exception when executing the try block. This is useful when we want to execute something only if there were no errors.

do not run, just to illustrate

```
# potential code before try catch

try:
    # code to try to execute
except ZeroDivisionError:
    # code to execute if there is a ZeroDivisionError
except NameError:
    # code to execute if there is a NameError
except:
    # code to execute if ther is any exception
else:
    # code to execute if there is no exception

# code that will execute if there is no exception or a one that we are
handling
```

finally allows us to always execute something even if there is an exception or not. This is usually used to signify the end of the try except.

```python
# potential code before try catch

try:
    # code to try to execute
except ZeroDivisionError:
    # code to execute if there is a ZeroDivisionError
except NameError:
    # code to execute if there is a NameError
except:
    # code to execute if ther is any exception
else:
    # code to execute if there is no exception
finally:
    # code to execute at the end of the try except no matter what

# code that will execute if there is no exception or a one that we are
handling
```

Try Except Else and Finally Example

You might have noticed that even if there is an error the value of a is always printed. Let's use the else and print the value of a only if there is no error.

```python
a = 1

try:
    b = int(input("Please enter a number to divide a"))
    a = a/b
except ZeroDivisionError:
    print("The number you provided cant divide 1 because it is 0")
except ValueError:
    print("You did not provide a number")
except:
    print("Something went wrong")
else:
    print("success a=",a)
```

Now lets let the user know that we are done processing their answer. Using the finally, let's add a print.

```python
a = 1

try:
    b = int(input("Please enter a number to divide a"))
    a = a/b
except ZeroDivisionError:
    print("The number you provided cant divide 1 because it is 0")
except ValueError:
    print("You did not provide a number")
```

```
except:
    print("Something went wrong")
else:
    print("success a=",a)
finally:
    print("Processing Complete")
```

Exercise 1: Handling ZeroDivisionError

Imagine you have two numbers and want to determine what happens when you divide one number by the other. To do this, you need to create a Python function called `safe_divide.` You give this function two numbers, a `'numerator'` and a `'denominator'`. The 'numerator' is the number you want to divide, and the `'denominator'` is the number you want to divide by. Use the user input method of Python to take the values.

The function should be able to do the division for you and give you the result. But here's the catch: if you try to divide by zero (which is not allowed in math), the function should be smart enough to catch that and tell you that it's not possible to divide by zero. Instead of showing an error, it should return None, which means 'nothing' or 'no value', and print `"Error: Cannot divide by Zero.`

```
#Type your code here
def safe_divide(numerator,denominator):
    try:
        result = numerator / denominator
        return result
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        return None
# Test case
numerator=int(input("Enter the numerator value:-"))
denominator=int(input("Enter the denominator value:-"))
print(safe_divide(numerator,denominator))
```

Note:- Practice handling exceptions by trying different input types like using integers, strings, zero, negative values, or other data types.

Exercise 2: Handling ValueError

Imagine you have a number and want to calculate its square root. To do this, you need to create a Python function. You give this function one number, `'number1'`.

The function should generate the square root value if you provide a positive integer or float value as input. However, the function should be clever enough to detect the mistake if you enter a negative value. It should kindly inform you with a message saying, `'Invalid input! Please enter a positive integer or a float value.`

```python
#Type your code here
import math

def perform_calculation(number1):
    try:
        result = math.sqrt(number1)
        print(f"Result: {result}")
    except ValueError:
        print("Error: Invalid input! Please enter a positive integer
or a float value.")
# Test case
number1=float(input("Enter the number:-"))
perform_calculation(number1)

#Note:- Test with different inputs to validate error handling.
```

Note:- Practice handling exceptions by trying different input types like using integers, strings, zero, negative values, or other data types.

Exercise 3: Handling Generic Exceptions

Imagine you have a number and want to perform a complex mathematical task. The calculation requires dividing the value of the input argument `"num"` by the difference between `"num"` and 5, and the result has to be stored in a variable called `"result"`.

You have to define a function so that it can perform that complex mathematical task. The function should handle any potential errors that occur during the calculation. To do this, you can use a try-except block. If any exception arises during the calculation, it should catch the error using the generic exception class `"Exception" as "e"`. When an exception occurs, the function should display `"An error occurred during calculation.`

```python
#Type your code here
def complex_calculation(num):
    try:
        result = num / (num - 5)
        print (f"Result: {result}")
    except Exception as e:
        print("An error occurred during calculation.")
# Test case
user_input = float(input("Enter a number: "))
complex_calculation(user_input)
```

Note:- Practice handling exceptions by trying different input types like using integers, strings, zero, negative values, or other data types.

Authors

Joseph Santarcangelo

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2023-11-02	2.2	Abhishek Gagneja	Updated instructions
2023-08-01	2.1	Akansha Yadav	Added optional practice section
2020-09-02	2.0	Simran	Template updates to the file