



Python

Experiment No. 10

Aim: To implement exception handling and file handling operations with exception handling.

Problem Statements:

1. Create a Python function that checks if a given number is prime. Handle exceptions for non-integer input and provide informative error messages.
2. You're going to write an interactive calculator! User input is assumed to be a formula that consist of a number, an operator (at least + and -), and another number, separated by white space (e.g. 1 + 1). Split user input using str.split(), and check whether the resulting list is valid:
 - a. If the input does not consist of 3 elements, raise a FormulaError, which is a custom Exception.
 - b. Try to convert the first and third input to a float (like so: float_value = float(str_value)). Catch any ValueError that occurs, and instead raise a FormulaError
 - c. If the second input is not '+' or '-', again raise a FormulaError

If the input is valid, perform the calculation and print out the result. The user is then prompted to provide new input, and so on, until the user enters quit.

3. Write a Python program that opens a file and display contents of file and handles a FileNotFoundError exception if the file does not exist.
4. Write a Python program that opens a file in write mode using with..open and handles a PermissionError exception if there is a permission issue.

Theory:

An **exception** is an unexpected event that occurs during program execution.

For example,

divide_by_zero = 7 / 0

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)
- try to import a module that does not exist (ImportError) and so on.

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.

Errors represent conditions such as compilation error, syntax error, error in the logical part of the code, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer and we should not try to handle errors.

Exceptions can be caught and handled by the program.

Handling an exception

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Python try...except Block

The try...except block is used to handle exceptions in Python. Here's the syntax of try...except block:

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

Example: Exception Handling Using try...except

Code:

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")
```

Output:

```
➞ Error: Denominator cannot be 0.
```

Catching Specific Exceptions in Python

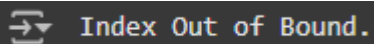
For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle each exception differently.

The argument type of each except block indicates the type of exception that can be handled by it. For example,

Code:

```
try:
    even_numbers = [2,4,6,8]
    print(even_numbers[5])
except ZeroDivisionError:
    print("Denominator cannot be 0.")
except IndexError:
    print("Index Out of Bound.")
```

Output:

A terminal window showing the output of the code. It displays a dark background with a light-colored icon on the left and the text "Index Out of Bound." in a light color.

Python try with else clause

In some situations, we might want to run a certain block of code if the code block inside try runs without any errors.

For these cases, you can use the optional else keyword with the try statement.

Let's look at an example:

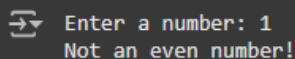
Code:

```
# program to print the reciprocal of even numbers

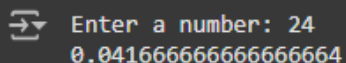
try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Output:

If we pass an odd number:

A terminal window showing the input "Enter a number: 1" and the output "Not an even number!". The background is dark, and the text is light-colored.

If we pass an even number, the reciprocal is computed and displayed:

A terminal window showing the input "Enter a number: 24" and the output "0.041666666666666664". The background is dark, and the text is light-colored.

However, if we pass 0, we get `ZeroDivisionError` as the code block inside `else` is not handled by preceding `except`:

```
Enter a number: 0
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-6-6e257cf75558> in <cell line: 3>()
      7     print("Not an even number!")
      8 else:
----> 9     reciprocal = 1/num
     10     print(reciprocal)

ZeroDivisionError: division by zero
```

Note: Exceptions in the `else` clause are not handled by the preceding `except` clauses.

Python try...finally

In Python, the `finally` block is always executed no matter whether there is an exception or not. The `finally` block is optional. And, for each `try` block, there can be only one `finally` block. Let's see an example,

Code:

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")
```

Output:

```
Error: Denominator cannot be 0.
This is finally block.
```

Defining Custom Exceptions

In Python, we can define custom exceptions by creating a new class that is derived from the built-in `Exception` class.

Here's the **syntax** to define custom exceptions,

```
class CustomError(Exception):
```

```
...
    pass
```

```
try:
    ...

except CustomError:
    ...
```

Example Code:

```
# define Python user-defined exceptions
class InvalidAgeException(Exception):
    "Raised when the input value is less than 18"
    pass

# you need to guess this number
number = 18

try:
    input_num = int(input("Enter a number: "))
    if input_num < number:
        raise InvalidAgeException
    else:
        print("Eligible to Vote")

except InvalidAgeException:
    print("Exception occurred: Invalid Age")
```

Output:

```
➡ Enter a number: 15
Exception occurred: Invalid Age
```

List of Standard Exceptions:

Sr.No.	Exception Name & Description
1	Exception Base class for all exceptions
2	StopIteration Raised when the next() method of an iterator does not point to any object.
3	SystemExit Raised by the sys.exit() function.

4	StandardError Base class for all built-in exceptions except StopIteration and SystemExit.
5	ArithmeticError Base class for all errors that occur for numeric calculation.
6	OverflowError Raised when a calculation exceeds maximum limit for a numeric type.
7	FloatingPointError Raised when a floating point calculation fails.
8	ZeroDivisionError Raised when division or modulo by zero takes place for all numeric types.
9	AssertionError Raised in case of failure of the Assert statement.
10	AttributeError Raised in case of failure of attribute reference or assignment.

Python File Operation

A file is a container in computer storage devices used for storing data.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

- Open a file
- Read or write (perform operation)
- Close the file

Opening Files in Python

In Python, we use the open() method to open files.

Let's suppose we have a file named test.txt with the following content.

```
# open file in current directory
file1 = open("test.txt")
```

By default, the files are open in read mode (cannot be modified). The code above is equivalent to

```
file1 = open("test.txt", "r")
```

Different Modes to Open a File in Python

Mode	Description
r	Open a file for reading. (default)
w	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Open a file for exclusive creation. If the file already exists, the operation fails.
a	Open a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Open in text mode. (default)
b	Open in binary mode.
+	Open a file for updating (reading and writing)

```
file1 = open("test.txt") # equivalent to 'r' or 'rt'
file1 = open("test.txt", 'w') # write in text mode
file1 = open("img.bmp", 'r+b') # read and write in binary mode
```

Reading Files in Python

After we open a file, we use the read() method to read its contents. For example,

```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)

#Output: Contents of the file
```

In the above example, we have read the file that is available in our current directory.

Notice the code,

```
read_content = file1.read
```

Here, file1.read() reads the test.txt file and is stored in the read_content variable.

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file. Closing a file will free up the resources that were tied with the file. It is done using the close() method in Python. For example,

```
# open a file
file1 = open("test.txt", "r")

# read the file
```

```
read_content = file1.read()
print(read_content)

# close the file
file1.close()
```

Exception Handling in Files

If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try...finally block.

Let's see an example,

```
try:
    file1 = open("test.txt", "r")
    read_content = file1.read()
    print(read_content)

finally:
    # close the file
    file1.close()
```

Here, we have closed the file in the finally block as finally always executes, and the file will be closed even if an exception occurs.

Use of with...open Syntax

In Python, we can use the with...open syntax to automatically close the file.

For example,

```
with open("test.txt", "r") as file1:
    read_content = file1.read()
    print(read_content)
```

Note: Since we don't have to worry about closing the file, make a habit of using the with...open syntax.

Writing to Files in Python

There are two things we need to remember while writing to a file.

- If we try to open a file that doesn't exist, a new file is created.
- If a file already exists, its content is erased, and new content is added to the file.

In order to write into a file in Python, we need to open it in write mode by passing "w" inside open() as a second argument.

Suppose, we don't have a file named test2.txt. Let's see what happens if we write contents to the test2.txt file.

```
with open(test2.txt, 'w') as file2:
```



```
# write contents to the test2.txt file
file2.write('Programming is Fun.')
fil2.write('Programiz for beginners')
```

Here, a new test2.txt file is created and this file will have contents specified inside the write() method.

Python Files Methods

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the TextIOBase and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns True if the file stream is interactive.
read(<u>n</u>)	Reads at most <u>n</u> characters from the file. Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(<u>n</u> =-1)	Reads and returns one line from the file. Reads in at most <u>n</u> bytes if specified.
readlines(<u>n</u> =-1)	Reads and returns a list of lines from the file. Reads in at most <u>n</u> bytes/characters if specified.
seek(<u>offset</u> , <u>from</u> =SEEK_SET)	Changes the file position to <u>offset</u> bytes, in reference to <u>from</u> (start, current, end).
seekable()	Returns True if the file stream supports random access.
tell()	Returns an integer that represents the current position of the file's object.
truncate(<u>size</u> =None)	Resizes the file stream to <u>size</u> bytes. If <u>size</u> is not specified, resizes to current location.
writable()	Returns True if the file stream can be written to.
write(<u>s</u>)	Writes the string <u>s</u> to the file and returns the number of characters written.
writelines(<u>lines</u>)	Writes a list of <u>lines</u> to the file.

Q.1 Create a Python function that checks if a given number is prime. Handle exceptions for non-integer input and provide informative error messages.

Code:

```
def isPrime(num):
    try:
        num = int(num)
        if num < 2:
            return False
        for i in range(2, int(num**0.5)+1):
            if num % i == 0:
                return False
        return True
    except ValueError:
        return ValueError("Input is not a valid Integer")

try:
    num = input("Enter a number: ")
    if isPrime(num):
        print(f"{num} is a prime number")
    else:
        print(f"{num} is not a prime number")
except ValueError as e:
    print(e)
```

Output:

```
Enter a number: 40
40 is not a prime number
```

Q.2 You're going to write an interactive calculator! User input is assumed to be a formula that consist of a number, an operator (at least + and -), and another number, separated by white space (e.g. 1 + 1). Split user input using `str.split()`, and check whether the resulting list is valid:

- a. If the input does not consist of 3 elements, raise a `FormulaError`, which is a custom Exception.
- b. Try to convert the first and third input to a float (like so: `float_value = float(str_value)`). Catch any `ValueError` that occurs, and instead raise a `FormulaError`
- c. If the second input is not '+' or '-', again raise a `FormulaError`

If the input is valid, perform the calculation and print out the result. The user is then prompted to provide new input, and so on, until the user enters quit.

Code:

```
class FormulaError(Exception):
    pass

def calculate(formula):
    elements = formula.split()
    if len(elements) != 3:
        raise FormulaError("Invalid formula. Expected
format: number operator number")
    try:
        num1=float(elements[0])
        num2=float(elements[2])
    except ValueError:
        raise FormulaError("Invalid formula. Numbers must be
valid numbers")

    if elements[1] not in ['+', '-']:
        raise FormulaError("Invalid formula. Operator must
be '+' or '-'")

    if elements[1] == '+':
        return num1 + num2
    else:
        return num1 - num2

while True:
    formula=input("Enter a formula(eg. 1+1) (or 'quit' to
exit): ")
    if formula.lower() == 'quit':
        break
    try:
        result=calculate(formula)
```

```
print("\nResult:", result)
except FormulaError as e:
    print("\nError:", e)
```

Output:

```
➡ Enter a formula(eg. 1+1) (or 'quit' to exit): 1 2 3
Error: Invalid formula. Operator must be '+' or '-'
Enter a formula(eg. 1+1) (or 'quit' to exit): 1 + 1

Result: 2.0
Enter a formula(eg. 1+1) (or 'quit' to exit): 2 + +

Error: Invalid formula. Numbers must be valid numbers
Enter a formula(eg. 1+1) (or 'quit' to exit): QUIT
```

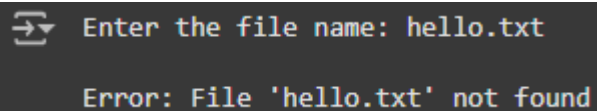
Q.3 Write a Python program that opens a file and display contents of file and handles a FileNotFoundError exception if the file does not exist.

Code:

```
try:
    file_name=input("Enter the file name: ")
    with open(file_name, 'r') as file:
        content=file.read()
        print("\nFile Content:")
        print(content)

except FileNotFoundError:
    print(f"\nError: File '{file_name}' not found")
except Exception as e:
    print(f"\nError: {e}")
```

Output:



```
➞ Enter the file name: hello.txt
Error: File 'hello.txt' not found
```

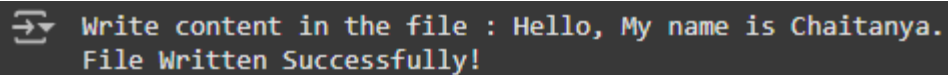
Q.4 Write a Python program that opens a file in write mode using with..open and handles a PermissionError exception if there is a permission issue.

Code:

```
try:
    with open("S009_Expt10_Text.txt",'w') as file:
        text=input("Write content in the file : ")
        print("File Written Successfully!")
        file.write(text)

except PermissionError:
    print("\nError: Permission denied to write to the file")
```

Output:



```
➔ Write content in the file : Hello, My name is Chaitanya.
File Written Successfully!
```

Conclusion: Thus studied exception and file handling

- Chaitanya Shah