Name: Chandana Ramesh Galgali
Batch: P6-1 Roll No.: 16010422234
Experiment / ~~assignment~~ / ~~tutorial~~ No. 8
Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Exception handling in Python

AIM: Write a program to demonstrate exception handling mechanism in Python

Expected OUTCOME of Experiment: To demonstrate how exceptions can be used to improve the robustness, reliability, and maintainability of our programs by handling unexpected errors effectively

Resource Needed: Python IDE

Theory:

What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Built-in Exceptions

Commonly occurring exceptions are usually defined in the compiler/interpreter. These are called built-in exceptions. Python's standard library is an extensive collection of built-in exceptions that deals with the commonly occurring errors (exceptions) by providing the standardized solutions for such errors. On the occurrence of any built-in exception, the appropriate exception handler code is executed which displays the reason along with the raised exception name. The programmer then has to take appropriate action to handle it. Some of the commonly occurring built-in exceptions that can be raised in Python are explained below

Following are list of common exception class found in Python:

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

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.

The try except else block:

Syntax:

```
try:
 You do your operations here;
except ExceptionI:
If there is ExceptionI, then execute this block
else:
If there is no Exception, then execute this block
```

Here are few important points about the above-mentioned syntax:

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example:

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
```

```
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

This produces the following result:

Written content in the file successfully

Example:

This example tries to open a file where you do not have write permission, so it raises an exception

```
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

This produces the following result:

Error: can't find file or read data

The except Clause with No Exceptions:

You can also use the except statement with no exceptions defined as follows:

```
try:
You do your operations here;
……..
except:
If there is any exception, then execute this block
…….
else
If there is no exception, then execute this block
…….
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The except Clause with Multiple Exceptions:

You can also use the same except statement to handle multiple exceptions as follows:

```
try:
```

```
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause:

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use else clause as well along with a finally clause.

Example

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result

Error: can't find file or read data

Same example can be written more cleanly as follows:

```
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

Argument of an Exception:

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception

```
# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError, Argument:
      print "The argument does not contain numbers\n", Argument

# Call above function here.
temp_convert("xyz");
```

This produces the following result:

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

Raising an Exceptions:

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

Syntax:

raise [Exception [, args [, traceback]]]

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows:

```
def functionName( level ):
  if level < 1:
    raise "Invalid level!", level
    # The code below to this would not be executed
    # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows −

```
try:
  Business Logic here...
except "Invalid level!":
  Exception handling here...
else:
  Rest of the code here...
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to RuntimeError. Here, a class is created that is subclassed from RuntimeError. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

```
class Networkerror(RuntimeError):
  def __init__(self, arg):
    self.args = arg
```

So once you defined above class, you can raise the exception as follows −

```
try:
  raise Networkerror("Bad hostname")
except Networkerror,e:
  print e.args
```

Problem Definition:

1. Write a program in which we prompt the user to enter personal details like name and surname should be strings, age should be an integer, height and weight should be float. Whenever the user enters input of the incorrect type, keep prompting the user for the same value until it is entered correctly. Give the user sensible feedback.

2. Write a program that prompts the user to enter a password and checks if it meets certain criteria (such as length, containing special characters, etc.). If the password does not meet the criteria, raise a custom exception InvalidPasswordError with an appropriate error message.

Books/ Journals/ Websites referred:

1. Reema Thareja, Python Programming: Using Problem Solving Approach, Oxford University Press, First Edition 2017, India
2. Sheetal Taneja and Naveen Kumar, Python Programming: A modular Approach, Pearson India, Second Edition 2018,India

---

Implementation details:

Q.1

```python
def get_input(prompt, input_type):
    while True:
        try:
            value = input_type(input(prompt))
            return value
        except ValueError:
            print(f"Invalid input. Please enter a {input_type.__name__}.")

def main():
    name = get_input("Enter your name: ", str)
```

```
    surname = get_input("Enter your surname: ", str)
    age = get_input("Enter your age: ", int)
    height = get_input("Enter your height (in meters): ", float)
    weight = get_input("Enter your weight (in kilograms): ", float)

    print(f"\nPersonal Details:\nName: {name} {surname}\nAge:
{age}\nHeight: {height} meters\nWeight: {weight} kilograms")

if __name__ == "__main__":
    main()
```

Q.2

```
import re

class InvalidPasswordError(Exception):
    pass

def check_password(password):
    if len(password) < 8:
        raise InvalidPasswordError("Password must be at least 8 characters
long.")
    if not re.search(r"[A-Z]", password):
        raise InvalidPasswordError("Password must contain at least one
uppercase letter.")
    if not re.search(r"[a-z]", password):
        raise InvalidPasswordError("Password must contain at least one
lowercase letter.")
    if not re.search(r"[0-9]", password):
        raise InvalidPasswordError("Password must contain at least one
digit.")
    if not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
        raise InvalidPasswordError("Password must contain at least one
special character.")
    return True

def main():
    password = input("Enter a password: ")
    try:
        check_password(password)
```

```
        print("Password is valid.")
    except InvalidPasswordError as e:
        print(f"Invalid password: {e}")

if __name__ == "__main__":
    main()
```

Output(s):

Q.1

```
Enter your name: Chandana
Enter your surname: Galgali
Enter your age: 18
Enter your height (in meters): 164
Enter your weight (in kilograms): 54

Personal Details:
Name: Chandana Galgali
Age: 18
Height: 164.0 meters
Weight: 54.0 kilograms
```

Q.2

```
Enter a password: Cndr@3904
Password is valid.
```

Conclusion:
Exception handling cases have been implemented and compiled.

Post Lab Questions:

1. Write a program that takes a list of numbers as input from the user and calculates their average. If the list is empty, raise a custom exception EmptyListError with an appropriate error message.
   Ans:

```
class EmptyListError(Exception):
    pass

def calculate_average(numbers):
    if not numbers:
```

```
        raise EmptyListError("List is empty.")
    return sum(numbers) / len(numbers)

def main():
    numbers = []
    while True:
        try:
            num = input("Enter a number (or 'done' to finish): ")
            if num == "done":
                break
            numbers.append(float(num))
        except ValueError:
            print("Invalid input. Please enter a number.")

    try:
        avg = calculate_average(numbers)
        print(f"Average: {avg}")
    except EmptyListError as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

```
Enter a number (or 'done' to finish): 7
Enter a number (or 'done' to finish): 3
Enter a number (or 'done' to finish): 8
Enter a number (or 'done' to finish): 4
Enter a number (or 'done' to finish): done
Average: 5.5
```

2. What is the purpose of the "finally" block in a try-except-finally block?

   Ans: The finally block in a try-except-finally block is used to define a set of statements that will be executed regardless of whether an exception was raised or not. The finally block is optional, but it's useful for performing cleanup operations, such as closing files or releasing resources, that need to be done regardless of whether an exception was raised or not.

Date: _____ **Signature of faculty in-charge**