Experiment 07

Write python programs to understand
7.1) Creating User-defined modules/packages and import them in a program
7.2) Creating a menu driven application which should cover all the built-in exceptions in python, also demonstrate the concept of assertion and user defined exception

Roll No.	61
Name	V Krishnasubramaniam
Class	D10-A
Subject	Python Lab
LO Mapped	LO1: Understand the structure, syntax, and semantics of the Python language LO4: Create Python applications using modules, packages, multithreading and exception handling.

Aim:

Write python programs to understand
7.1) Creating User-defined modules/packages and import them in a program
7.2) Creating a menu driven application which should cover all the built-in exceptions in python, also demonstrate the concept of assertion and user defined exception

Introduction:

Modules in Python

A module is a Python object with arbitrarily named attributes that you can bind and reference. Simply, a module is a file consisting of Python code. Any text file with the .py extension containing Python code is basically a module. A module can define functions, classes and variables. A module can also include runnable code. Definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode). Within a module, the module's name (as a string) is available as the value of the global variable __name__.

Example:
Code in mypackage1.py:

```
def sum(x, y):
        return x + y
```

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax −

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. We can use as keyword to rename the module we are importing. Every module, either built-in or custom made, is an object of a module class. Verify the type of different modules using the built-in type() function, as shown below.

Example:

```
import mypackage1 as pk
pk.sum(3,10)
print(type(pk))
```

Output:

```
13
```

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax −

```
from modname import name1[, name2[, ... nameN]]
```

This statement does not import the entire module fib into the current namespace; it just introduces the specified item from the module into the global symbol table of the importing module.

Example:
Code in mypackage2.py:

```
def sum(x, y):
        return x + y
def multiply(x,y):
return x*y
def subtract(x,y):
return x-y
```

Code in run2.py:

```
from mypackage2 import sum, subtract
sum(3,10)
subtract(10,3)
```

Output:

```
13
7
```

Module Search Path:

When you import a module, the Python interpreter searches for the module in the following sequences −

1. The current directory.
2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
3. If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

Example:

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\python39.zip',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\DLLs',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\lib',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39',
'C:\\Users\\vkris\\AppData\\Roaming\\Python\\Python39\\site-packages',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\lib\\site-packages',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\lib\\site-packages\\win32',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\lib\\site-packages\\win32\\lib',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\lib\\site-packages\\Pythonwin']
```

PYTHONPATH variable:

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH. Here is a typical PYTHONPATH from a Windows system −

set PYTHONPATH = c:\python20\lib;

If the required module is not present in any of the directories above, the error message ModuleNotFoundError is thrown.

Reloading a module:
Suppose you have already imported a module and using it. However, the owner of the module added or modified some functionalities after you imported it. So, you can reload the module to get the latest module using the reload() function of the imp module.

Example:
```
import imp
imp.reload(mypackage1)
```

Standard Modules:
Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform.

Example:
```
import sys
sys.ps1
```

Output:
```
'>>> '
```

We can use the dir() function to find out names that are defined inside a module.

Syntax:
```
dir(moduleName)
```

Python re module

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern. RegEx can be used to check if a string contains the specified search pattern. Python has a built-in package called re, which can be used to work with Regular Expressions. When you have imported the re module, we can use the functions, metacharacters and sets to search, find or extract substrings from a string.

Example:
```
import re
 txt = "The rain in Spain"
x = re.findall("^The.*Spain$", txt)
x                                    #Output: ['The rain in Spain']
```

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Packages are a way of structuring Python's module namespace by using "dotted module names". A package in Python takes the concept of the modular approach to next logical level. As you know, a module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files.

Creating a package is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. Here, there is a directory named pkg that contains two modules, mod1.py and mod2.py. The contents of the modules are:

Code in mod1.py:

```
def foo():
        print('[mod1] foo()')
class Foo:
        pass
```

Code in mod2.py:

```
def bar():
        print('[mod2] bar()')
class Bar:
        pass
```

Given this structure, if the pkg directory resides in a location where it can be found (in one of the directories contained in sys.path), you can refer to the two modules with dot notation (pkg.mod1, pkg.mod2) and import them with the syntax :

import package_name.<module_name>[, package_name. <module_name> ...]

Example:

```
>>> import pkg.mod1, pkg.mod2
>>> pkg.mod1.foo()                  #Output: [mod1] foo()
>>> x = pkg.mod2.Bar()
>>> x                               #Output: <pkg.mod2.Bar object at 0x033F7290>
```

If a file named __init__.py is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

Example:

Code in __init__.py:

```
print(f'Invoking __init__.py for {__name__}')
A = ['quux', 'corge', 'grault']
```

A module in the package can access the global variable by importing it in turn:

Code in mod1.py:

```
def foo():
        from pkg import A
        print('[mod1] foo() / A = ', A)
class Foo:
        pass
```

```
>>> from pkg import mod1
>>> mod1.foo()                    #Output: [mod1] foo() / A =  ['quux', 'corge', 'grault']
```

Exceptions in Python

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 can't cope with, it raises an exception.

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program. 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 would terminate and come out.

Exception Handling: 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 statements to handle errors and exceptions in Python are as follows:

1. try/except: Catch and recover from raised by you or Python exceptions
2. try/finally: Perform cleanup actions whether exceptions occur or not
3. raise: Trigger an exception manually in your code
4. assert: Conditionally trigger an exception in your code

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.

Syntax:

```
try:
You do your operations here;
except ExceptionI:
If there is ExceptionI, then execute this block.
```

except ExceptionII:
If there is ExceptionII, then execute this block.
else:
If there is no exception then execute this block.

First the 'try' clause is executed until an exception occurs, in which case the rest of the 'try' clause is skipped and the 'except' clause is executed (depending on type of exception), and execution continues. If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements. If no handler is found, it is an unhandled exception and execution stops.

The last except clause (when many are declared) may omit the exception name(s), to serve as a wildcard. This makes it very easy to mask a real programming error. It can also be used to print an error message and then re-raise the exception.

Except clause with no exceptions: You can also use the except statement with no exceptions defined as the following:

Syntax:
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.

Example:

```
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

Output:

```
The entry is a
Oops! <class 'ValueError'> occurred.
```

Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5

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

Syntax:
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.

Example:

```
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except Exception as e:
        print("Oops!", e.__class__, "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

Output:
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5

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.

Syntax:
try:
You do your operations here;
finally:
This would always be executed.

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

Example code snippet:

```
try:
  f = open("test.txt",encoding = 'utf-8')
  #perform file operations
finally:
  f.close()
```

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.

Syntax:
try:
You do your operations here;
except ExceptionType, Argument:
You can print the 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.

Raising an exception: 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 code snippet:

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

Examples of Exceptions

1. Division by Zero
2. Addition of two incompatible types
3. Accessing a file that is nonexistent.
4. Accessing a nonexistent index of a sequence.
5. Deleting a table in a disconnected database server.
6. Withdrawing money greater than the available amount.

Built-in Exceptions

In Python, all exceptions must be instances of a class that derives from BaseException. In a try statement with an except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the Exception class or one of its subclasses, and not from BaseException.

Some of the common built-in exceptions are:

1. exception **AssertionError**: Raised when an assert statement fails.
2. exception **AttributeError**: Raised when an attribute reference or assignment fails. (When an object does not support attribute references or attribute assignments at all, TypeError is raised.)
3. exception **EOFError**: Raised when the input() function hits an end-of-file condition (EOF) without reading any data.
4. exception **GeneratorExit**: Raised when a generator or coroutine is closed. It directly inherits from BaseException instead of Exception since it is technically not an error.
5. exception **ImportError**: Raised when the import statement has troubles trying to load a module. Also raised when the “from list” in from ... import has a name that cannot be found. The name and path attributes can be set using keyword-only arguments to the constructor. When set they represent the name of the module that was attempted to be imported and the path to any file which triggered the exception, respectively.

6. exception **ModuleNotFoundError**: A subclass of ImportError which is raised by import when a module could not be located. It is also raised when None is found in sys.modules.
7. exception **IndexError**: Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, TypeError is raised.)
8. exception **KeyError**: Raised when a mapping (dictionary) key is not found in the set of existing keys.
9. exception **KeyboardInterrupt**: Raised when the user hits the interrupt key. During execution, a check for interrupts is made regularly. The exception inherits from BaseException so as to not be accidentally caught by code that catches Exception and thus prevent the interpreter from exiting.
10. exception **MemoryError**: Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory.
11. exception **NameError**: Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.
12. exception **NotImplementedError**: This exception is derived from RuntimeError. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

User-defined Exceptions

In Python, users can define custom exceptions by creating a new class. Users can create exception classes to handle exceptions specific to a program. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class.

Here we created a new exception class i.e. User_Error. Exceptions need to be derived from the built-in Exception class, either directly or indirectly. Let's look at the given example which contains a constructor and display method within the given class.

Example:

```
class User_Error(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return(repr(self.value))
try:
  raise(User_Error("User defined error"))
except User_Error as error:
  print('A New Exception occured:',error.value)
```

Output:

A New Exception occured: User defined error

assert statement:

Assertion is a sanity-check that you can turn on/off when you are done with your testing of the program It's like a raise-if statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false an exception is raised.

Example:

```
x = "hello"
assert x == "hello"
assert x == "goodbye"
```

Output:

```
Traceback (most recent call last):
File "demo_ref_keyword_assert.py", line 5, in
<module>
assert x == "goodbye"
AssertionError
```

We can use this mechanism to handle custom user defined exceptions, by using except block to handle the AssertionError.

Example:

```
try:
    age= -10
    print("Age is:")
    print(age)
    assert age>0
    yearOfBirth= 2021-age
    print("Year of Birth is:")
    print(yearOfBirth)
except AssertionError:
    print("Input Correct age.")
```

Output:

```
Age is:
-10
Input Correct age.
```

Results:

Program in mymodule1.py:

```
def sum(x,y):
  return x+y
def subtract(x,y):
  return x-y
def multiply(x,y):
  return x*y
```

Program in run1.py:

```
import mymodule1 as md
print(md.sum(10,3))
print(md.subtract(10,3))
print(md.multiply(10,3))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py run1.py
13
7
30
```

Program in mymodule2.py:

```
def greeting(name):
  print("Hello, " + name)
person1 = {
  "name": "John Walker",
  "age": 36,
  "country": "USA"
}
```

Program in run1.py:

```
from mymodule2 import person1
print (person1["age"])
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py run2.py
36
```

Examples of preinstalled modules:

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\python39.zip',
'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Python39\\DLLs', 'C:\\Users\\v
kris\\AppData\\Local\\Programs\\Python\\Python39\\lib', 'C:\\Users\\vkris\\AppData\
\Local\\Programs\\Python\\Python39', 'C:\\Users\\vkris\\AppData\\Roaming\\Python\\P
ython39\\site-packages', 'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Pytho
n39\\lib\\site-packages', 'C:\\Users\\vkris\\AppData\\Local\\Programs\\Python\\Pyth
on39\\lib\\site-packages\\win32', 'C:\\Users\\vkris\\AppData\\Local\\Programs\\Pyth
on\\Python39\\lib\\site-packages\\win32\\lib', 'C:\\Users\\vkris\\AppData\\Local\\P
rograms\\Python\\Python39\\lib\\site-packages\\Pythonwin']
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
```

```
>>> import re
>>> txt = "The rain in Spain"
>>> x = re.findall("^The.*Spain$", txt)
>>> x
['The rain in Spain']
```

Modules in mypackage1:

Program in mod1.py:

```
def foo():
print('[mod1] foo()')
class Foo:
pass
```

Program in mod2.py:

```
def bar():
        print('[mod2] bar()')
class Bar:
        pass
```

Program in run3.py:

```
import mypackage1.mod1 as m1, mypackage1.mod2 as m2
m1.foo()
obj1 = m1.Foo()
print(obj1)
m2.bar()
obj2 = m2.Bar()
print(obj2)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py run3.py
[mod1] foo()
<mypackage1.mod1.Foo object at 0x00000203F617F5E0>
[mod2] bar()
<mypackage1.mod2.Bar object at 0x00000203F638E4F0>
```

Modules in package directory Cars:

Program in __init__.py:

```
import Cars.Bmw, Cars.Audi, Cars.Nissan
```

Program in Bmw.py:

```
class Bmw:
   def __init__(self):
      self.models = ['i8', 'x1', 'x5', 'x6']
   def outModels(self):
```

```
        print('These are the available models for BMW')
        for model in self.models:
            print('\t%s ' % model)
```

Program in Audi.py:

```
class Audi:
    def __init__(self):
        self.models = ['q7', 'a6', 'a8', 'a3']
    def outModels(self):
        print('These are the available models for Audi')
        for model in self.models:
            print('\t%s ' % model)
```

Program in Nissan.py:

```
class Nissan:
    def __init__(self):
        self.models = ['altima', '370z', 'cube', 'rogue']
    def outModels(self):
        print('These are the available models for Nissan')
        for model in self.models:
            print('\t%s ' % model)
```

Outside package:

Program in run4.py:

```
from Cars import Bmw, Audi,  Nissan
# Create an object of Bmw class & call its method
ModBMW = Bmw.Bmw()
ModBMW.outModels()
# Create an object of Audi class & call its method
ModAudi = Audi.Audi()
ModAudi.outModels()
# Create an object of Nissan class & call its method
ModNissan = Nissan.Nissan()
ModNissan.outModels()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py run4.py
These are the available models for BMW
        i8
        x1
        x5
        x6
These are the available models for Audi
        q7
        a6
        a8
        a3
These are the available models for Nissan
        altima
        370z
        cube
        rogue
```

Program in exception1.py:

```
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception1.py
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occurred.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

Program in exception2.py:

```
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except Exception as e:
        print("Oops!", e.__class__, "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception2.py
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occurred.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

Program in exception3.py:

```
class User_Error(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return(repr(self.value))
try:
  raise(User_Error("User defined error"))
except User_Error as error:
  print('A New Exception occured:',error.value)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception3.py
A New Exception occured: User defined error
```

Program in exception4.py:

```
class NegativeAgeError(Exception):
    pass
try:
    age= -10
```

```
    print("Age is:")
    print(age)
    if age<0:
        raise NegativeAgeError
    yearOfBirth= 2021-age
    print("Year of Birth is:")
    print(yearOfBirth)
except NegativeAgeError:
    print("Input Correct age.")
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception4.py
Age is:
-10
Input Correct age.
```

Program in exception5.py:

```
try:
    age= -10
    print("Age is:")
    print(age)
    assert age>0
    yearOfBirth= 2021-age
    print("Year of Birth is:")
    print(yearOfBirth)
except AssertionError:
    print("Input Correct age.")
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception5.py
Age is:
-10
Input Correct age.
```

Program in exception6.py:

```
def avg(marks):
    try:
        assert len(marks) != 0
        return sum(marks)/len(marks)
    except AssertionError:
        return "Empty list"

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
```

```
print("Average of mark1:",avg(mark1))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception6.py
Average of mark2: 78.0
Average of mark1: Empty list
```

Program in exception7.py:

```
class FractionalQuotientError(Exception):
  pass
while(1):
  try:
    y = 100
    s = int(input("Enter a number to divide 100: "))
    assert s>=0
    z = y/s
    if (z-int(z)) != 0:
      raise FractionalQuotientError
    print("Quotient: ",z)
  except ValueError as arg:
    print("Invalid Input!",arg)
  except ArithmeticError as arg:
    print("Invalid Input!",arg)
  except FloatingPointError as arg:
    print("An error occurred while dividing!",arg)
  except AssertionError as arg:
    print("Negative Number!",arg)
  except FractionalQuotientError as arg:
    print("Quotient is fractional!")
  else:
    print("Exit loop!")
    break
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception7.py
Enter a number to divide 100: 0
Invalid Input! division by zero
Enter a number to divide 100: -2
Negative Number!
Enter a number to divide 100: 8
Quotient is fractional!
Enter a number to divide 100: a
Invalid Input! invalid literal for int() with base 10: 'a'
Enter a number to divide 100: 5
Quotient:  20.0
Exit loop!
```

Program in exception8.py:

```
def assertionError(n1, n2):
    try:
        assert n2 != 0, "Invalid Operation"
        print(n1 / n2)
    except AssertionError as msg:
        print(msg)
def importError():
    try:
        from mypackage import greet
    except Exception as e:
        print(e)
def indexError(index):
    try:
        arr = [5, 10, 15, 20, 25]
        print("Size of array is 5")
        print(arr[index])
    except IndexError as e:
        print(e)
def keyError(key):
    try:
        dict = {"one": "Lorem", "two": "Ipsum", "three": "Dolor", "four": "Sit"}
        print("Dictionary have 4 keys (from one to four)")
        print(dict[key])
    except KeyError as e:
        print("Key not found",e)
def nameError():
    try:
        knownVariable = 20
        print(knownVariable)
        print(unknownVariable)
    except NameError as e:
        print(e)
```

```
def overflowError():
    i=1
    try:
        f = 3.0**i
        for i in range(100):
            print(i, f)
            f = f ** 2
    except OverflowError as err:
        print('Overflowed after ', f, err)
def syntaxError():
    try:
        print('Valid syntax')
        print(eval('*Not Valid Syntax*'))

    except SyntaxError as err:
        print (err)
def valueError():
    try:
        print (float('Krishna'))
    except ValueError as e:
        print (e)
def zeroDivisionError(n):
    try:
        print(20/n)

    except ZeroDivisionError as msg:
        print(msg)
choice = 0
while (True):
    print("""
    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT
    """)
    choice = int(input("Enter your choice :  "))

    if (choice == 1):
        print("Enter two numbers to divide")
        n1 = int(input("Enter number 1 :  "))
        n2 = int(input("Enter number 2 :  "))
        assertionError(n1, n2)
    elif (choice == 2):
        importError()
```

```
    elif (choice == 3):
        index = int(input("Enter an index (between 0 to 4) :  "))
        indexError(index)
    elif (choice == 4):
        key = input("Enter a key (between `one` to `four`) :  ")
        keyError(key)
    elif (choice == 5):
        nameError()
    elif (choice == 6):
        overflowError()
    elif (choice == 7):
        syntaxError()
    elif (choice == 8):
        valueError()
    elif (choice == 9):
        n = int(input("Enter a number to divide from :  "))
        zeroDivisionError(n)
    elif (choice == 10):
        print("EXITING.... ")
        break;
    else:
        print("Invalid choice")
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments\Exp 7>py exception8.py

    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  1
Enter two numbers to divide
Enter number 1 :  7
Enter number 2 :  0
Invalid Operation
```

```
    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  2
No module named 'mypackage'

    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT


Enter your choice :  3
Enter an index (between 0 to 4) :  5
Size of array is 5
list index out of range

    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  4
Enter a key (between `one` to `four`) :  3
Dictionary have 4 keys (from one to four)
Key not found '3'

    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  5
20
name 'unknownVariable' is not defined
```

```
    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  6
0 3.0
1 9.0
2 81.0
3 6561.0
4 43046721.0
5 1853020188851841.0
6 3.4336838202925124e+30
7 1.1790184577738583e+61
8 1.3900845237714473e+122
9 1.9323349832288915e+244
Overflowed after  1.9323349832288915e+244 (34, 'Result too large')


    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  7
Valid syntax
invalid syntax (<string>, line 1)

    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  8
could not convert string to float: 'Krishna'
```

```
Enter your choice :  9
Enter a number to divide from :  0
division by zero

    1. AssertionError       6.  OverflowError
    2. ImportError          7.  SyntaxError
    3. IndexError           8.  ValueError
    4. KeyError             9.  ZeroDivisionError
    5. NameError            10. EXIT

Enter your choice :  10
EXITING....
```

Conclusion:

Thus, we have understood the basics of modules and packages in Python, and how they help in the modularization of our programs. We also learnt about how Python handles Errors and Exceptions via in-built classes, and how we can create our custom, user-defined exceptions. We also performed hands-on practical programs for all of these concepts.