Statistical Analysis Using Python (MCSR140C)

Introduction to Python

By Malay Mitra

Day 1 and 2 / 7 Aug, 14 Aug 2023

- Python is a very simple language with straightforward syntax. It is case sensitive language.
- There are two major Python versions, Python 2 and Python 3. Python 2 and 3 are quite different. We would cover Python 3, because it is more semantically correct and supports newer features.
- For example, one difference between Python 2 and 3 is the print statement. In Python 2, the "print" statement is not a function, and therefore it is invoked without parentheses. However, in Python 3, it is a function, and must be invoked with parentheses.
- Python programs have extension .py

- To print a string in Python 3, just write:

```
# This is a comment line
print("This line will be printed.")                    # This is also a comment line
# Use the "print" command to print the line "Hello, World!".
print("Hello, World!")
```

- We can also use multiline comment as well.
  For example,

```
'''This is
multiline comment
in Python using triple quotes'''
```

- **Variables and Types**

Python is completely object oriented, and it is not "statically typed". You do not need to declare variables before using them in your Python program. Every variable in Python is treated as an object.

- Data types specify the type of data like numbers and characters to be stored and manipulated
  within a program. Basic data types of Python are

- Numbers
- Strings
- Boolean
- None

- **Numbers**

Python supports two types of numbers – integers, floating point and complex numbers
To define an integer, we use the following syntax:

```
intVar = 13
print(intVar)
```

To create a floating point number, we write:

```
floatVar = 71.0
print(floatVar)
myfloat = float(75)     # Or type cast integer to float
print(myfloat)
```

Similarly, we can convert float to int

```
x=90.55
y=int(x)
print(y)         # Output shows 90
```

```
# Also variables are case sensitive. x and X are different variables
x=90.55
X="python"
print(x,X)     #  Output shows 90.55 python
```

- **String data type**

Strings are defined either with a single quote or a double quote.

```
string1 = 'hello in single quote'
print(string1)
string2 = "hello in double quote”
print(string2)
```

When using single quote, it has to be paired with single quote. Similarly for double quote. Using both double and single quotes has some advantage as follows:

```
s = "Don't worry about Python"
print(s)      # Output “Don’t worry about Python”
```

- In Python, one can do basic mathematical calculations as follows:

```
>>> 5+6
11
>>> 7*2
14
>>> 5**2
25
>>> 4*4+5
21
```

- # We can do mathematical operations involving variables as mentioned below

```
var1 = 14
var2 = 21
var3 = var1 + var2
print(var3)      # Output 35
```

```
# String additions / concatenations in Python
>>> s1="hello"
>>> s2="world"
>>> s3=s1 + " " + s2
>>> print(s3)    # Output shows 'hello world'
```

- **Boolean data types**

Booleans may not seem to be very useful at first, but they are essential when we start using conditional statements in Python programming. A condition is really just a yes-or-no question, the answer to that question is a Boolean value, either True or False.

The Boolean values are True and False and they are treated as reserved words.

Example:

```
2>10     # False
10>2    #  True
a=19
b=100
z = a>b    # z becomes a Boolean variable and here its value is False
print(z)     # Output False
```

- **None data types**

None is another special data type in Python. *None* is frequently used to represent the absence of a value.

For example,

```
phoneNo = None
```

None value is assigned to variable phoneNo

- Checking the data types of a variable using 'type'

```
a = 10
b = 12.56
c = 'Python and Data Science'
d = None
e = a > 100
print(type(a), type(b), type(c), type(d), type(e))
```

Output of the above command shows

```
<class 'int'> <class 'float'> <class 'str'> <class 'NoneType'> <class 'bool'>
```

More examples of 'type' command

```
type(1)                          #   Returns <class 'int'>
type(6.4)                        #   Returns <class 'float'>
type("A")                        #   Returns <class 'str'>
type(True)                       #   Returns <class 'bool'>
```

- **Simultaneous assignments on more than one variable on the same line**

```
a, b = 5, 4
print(a,b)       # Output 5 4
# And then swap them without using any multiple lines code
a,b = b,a
print(a,b)             # Output 4 5
```

- **‘input’ function**

In Python, *input()* function is used to gather data from the user. The syntax for input function is

variable_name = input([prompt])   where prompt message is optional

Example:

```
age = input("What is your age?")
print(age)
print(type(age))          # Shows as class ‘str’
```

But here ‘age’ variable is of type string because ‘input’ function by default makes the data as string type. To convert the entered data into your required format we use type casting as given below.

```
age = int(input("What is your age?"))
print(age)
print(type(age))          # Shows as class ‘int’
```

- **Formatted output with “print” function**

```
# Program to Demonstrate input() and print() Functions
country = input("Which country do you live in?")
print("I live in {0}".format(country))
```

Output

```
Which country do you live in? India   (assume user types in “India” from keyboard)
I live in India
```

The 0 inside the curly braces {0} is the index of the first (0th) argument (here in our case, it is variable country) whose value will be inserted at that position.

```
#Program to Demonstrate the Positional Change of Indexes of Arguments
a = 10
b = 20
print("The values of a is {0} and b is {1}".format(a, b))     # variable ‘a’ goes as {0} and ‘b’ as {1}
print("The values of b is {1} and a is {0}".format(a, b))
```

Output

```
The values of a is 10 and b is 20
The values of b is 20 and a is 10
```

# **Print with ‘%’ and ‘%f ‘ format. %i stands for integer, %.2f stands for 2 decimal places**

```
a, b,c =10, 20.55, 30.678
print('%i, %.2f, %.4f' % (a, b, c))          # Outputs 10, 20.55, 30.6780
```

```
# Another easy way to print using f-string (or Formatted strings) as given below
min_temp = 21
max_temp = 31
print(f"Minimum temp :{min_temp}, Maximum temp: {max_temp}")
```

- **Operator “is” , “is not”, “in”, “not in”**

The operators *is* and *is not* are identity operators. Operator *is* evaluates to *True* if the values of operands on either side of the operator point to the same object and *False* otherwise.

Example:

```
x = 'Python'
y = 'Python'
print(x is y)                # Returns True as both them point to same object “Python”
print(x is not y)                     # Returns False as x and y point to same object

z = 'Python is good'
print(x is z)                # Returns False as ‘x’ and ‘z’ do not point to same object
print(x in z)                # But this returns True as string x is a substring in z
print(x not in z)                     # False
```

- List of Assignment Operators in Python

Operator	**Operator Name**	**Description**	**Example**	

=	Assignment	Assigns values from right side operands to left side operand.	z = p + q assigns value of p + q to z	x=15
				y=4
				z = x+y #19

+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand.	z += p is equivalent to z = z + p	x += y x becomes 19

−=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand.	z −= p is equivalent to z = z – p	x = 19
				y = 4
				x -= y
				x becomes 15

*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand.	z *= p is equivalent to z = z * p	x = 15
				y = 4
				x *= y
				x becomes 60

/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand.	z /= p is equivalent to z = z / p	x=21
				y=4
				x /= y
				x becomes 5.25

**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand.	z**= p is equivalent to z = z ** p	x=4
				y=3
				x **= y
				x becomes 64

//=	Floor Division Assignment	Produces the integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	z //= p is equivalent to z = z // p	x=23
				y=4
				x = x//y
				x becomes 5

%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	z %= p is equivalent to z = z % p	x=23
				y=4
				x = x%y
				x becomes 3

- List of Comparison operators.

Operator	**Operator Name**	**Description**	**Example. .** Consider p is 10 and q is 20.
==	Equal to	If the values of two operands are equal, then the condition becomes True	(p == q) is not True. That is False
!=	Not Equal to	If values of two operands are not equal, then the condition becomes True	(p != q) is True
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes True.	(p > q) is not True. That this False
<	Lesser than	If the value of left operand is less than the value of right operand, then condition becomes True.	(p < q) is True.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes True.	(p >= q) is not True. That is False
<=	Lesser than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes True.	(p <= q) is True.

- Logical Operators

Python supports three logical operators ‘and’, ‘or’ ‘not’

Operator	**Operator Name**	**Description**	**Example. Consider p=True and q=False**
and	Logical AND	Performs AND operation and the result is True when both operands are True	p and q is False
or	Logical OR	Performs OR operation and the result is True when any one of both operand is True	p or q is True
not	Logical NOT	Reverses the operand state	not p results in False not q is True

Some more examples

```
>>> 1 > 2 and 9 > 6     → False
>>> 3 > 2 or 8 < 4      → True
```

- List of Keywords in Python

and
assert
break
class
continue
def
del
elif
else
except
False

as
finally
for
from
global
if
import
in
is
lambda
True

not
or
pass
nonlocal
raise
return
try
while
with
yield
None

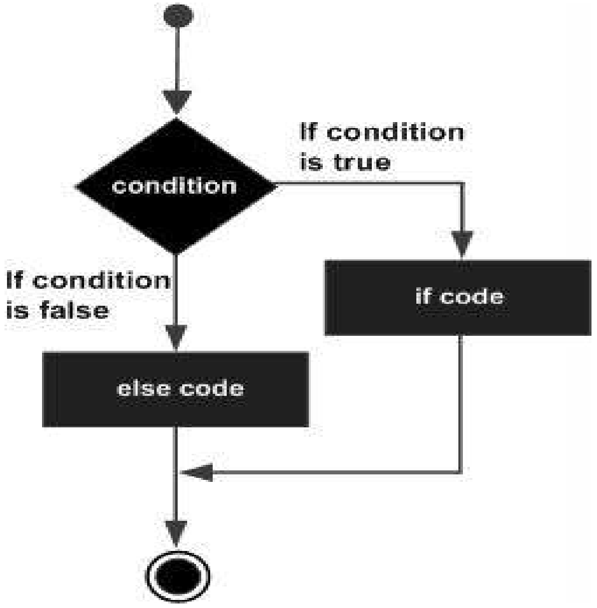
- **Control flow statements in Python**

If-elif-else block

In Python, the *if* block statements are determined through indentation and the first unindented statement marks the end. Brackets are optional.

Syntax:

```
if (Boolean condition1):
   Statements
elif(Boolean condition2):
   Statements
elif(Boolean condition3):
   Statements
else:
   Last Statements
```



Examples:

a)

```
if 20 > 10:
      print("20 is > 10")          #  Note the indentation

 Output: 20 is > 10
```

b)

```
number = int(input("Enter a number"))
if number >= 0:
  print("number positive")
else:
  print('number is negative')
Output:
      Enter a number 45
      number positive
```

c)

```
number = int(input("Enter a number"))
if number % 2 == 0:
      print(f"{number} is Even number")
else:
       print(f"{number} is Odd number")
```

Output

```
 Enter a number: 45
45 is Odd number
```

d)

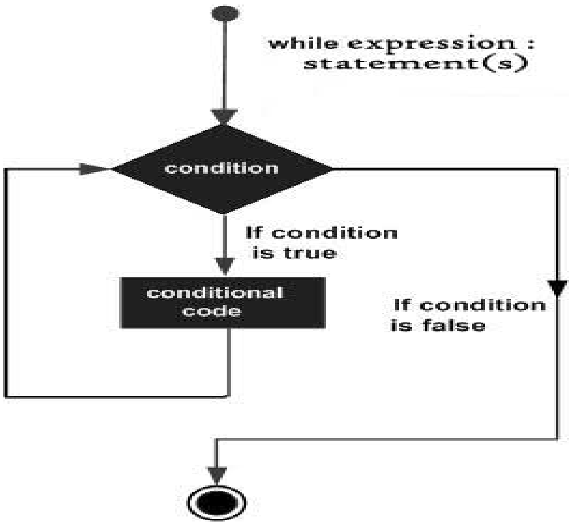
```
score = float(input("Enter your score"))
if score < 0 or score > 1:
   print('Wrong Input')
elif score >= 0.9:
      print('Your Grade is "A" ')
elif score >= 0.8:
       print('Your Grade is "B" ')
elif score >= 0.7:
        print('Your Grade is "C" ')
    elif score >= 0.6:
      print('Your Grade is "D" ')
    else:
      print('Your Grade is "F" ')
```

<u>The while loop</u>

There would be indentation in the while block

<u>Syntax:</u>

```
     while Boolean_Expression:
        statement(s)
```



Examples:

a)

```
# Python Program to Display First 10 natural numbers using
# while loop
i = 0
while i < 10:
        print(f"Current value of i is {i}")
        i = i + 1
```

Output:

```
Current value of i is 0
Current value of i is 1
Current value of i is 2
Current value of i is 3
Current value of i is 4
Current value of i is 5
Current value of i is 6
Current value of i is 7
Current value of i is 8
Current value of i is 9
```

b)

# Write a Program to Find the Average of *n* Natural Numbers
# Where *n is input f*rom the user

```
number = int(input("Enter a number up to which you want to find the average"))
i = 0
sum = 0
count = 0

while i < number:
        i = i + 1
        sum = sum + i
        count = count + 1     # This is the last statement in the while block

average = sum/count
print(f”The sum is {sum} : {sum}”)
print(f"The average of {number} natural numbers is {average}")
```

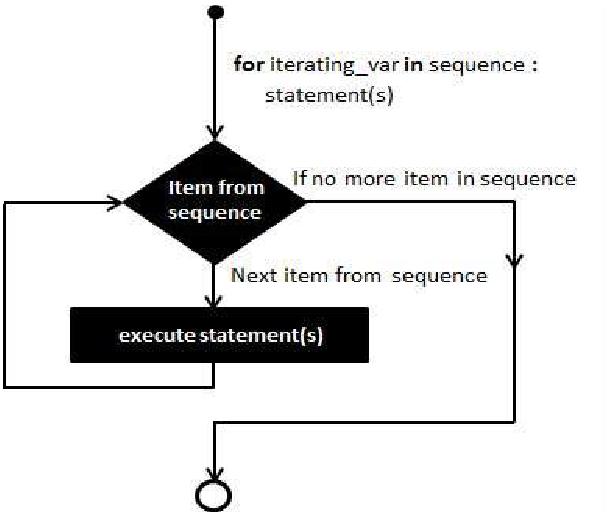
c)

```
capital = ' '
while capital.upper() != 'DELHI':
   capital = input('Enter our capital name :')
print('Got it')
```

The for Loop

Syntax:

```
for iteration_variable in sequence:
            statements
```



Note → Here sequence can be Tuple, List, Range etc. which would be covered next

Examples:

a)

```
# Example of for loop with Tuple data type. Any sequence within “( )” is a tuple
for i in (1,2,3):
        print(i)                # Output 1 2 3 in three lines. And (1,2,3) is called a tuple in Python
```

b)

```
# Similarly for loop with "list" data type. Any sequence within "[ ]" is a list
for i in [1,2,3]
        print(i)                # output 1 2 3 in three lines
```

c)

```
for ch in 'Python':
        print(ch)                    # Prints each character of the string vertically
```

- ***'range' function***

The *range()* function generates a sequence of numbers which can be iterated through using *for* loop. The syntax for *range()* function is

range([start ,] stop [, step])

Both start and step arguments are optional and the range argument value should always be an integer.
start → value indicates the beginning of the sequence. If the start argument is not specified, then it takes zero by default.
stop → Generates numbers up to this value but not including the number itself.
step → indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

NOTE: The square brackets in the syntax indicate that these arguments are optional. You can leave them out.

range(10)   → generates numbers from 0 to 9
range(0,10,2) → generates even numbers 2,4,6,8 between 0 to 9
range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

- **for loop with range function**

a)

```
for i in range(100):
        print i              # print the numbers from 0 through 99
```

b)

```
for i in range(2, 5):          # starts at 2, stops at 4, step is not mentioned it is 1 by default
        print(f"{i}")       # Prints 2, 3, 4
```

c)

```
for i in range(1, 6, 3):     # starts at 1, step 3, stops at 5
        print(f"{i}")       # Prints 1, 4
```

d)

```
# Reverse the numbers
for i in range(10, 0, -1):             # Step -1 means travel backwards. Starts at 10
        print(i)                       # Prints 10,9,8,………1
```

e)

```
# Write a Python program to find the sum of all odd and even numbers
#  up to a number as specified by the user.
number = int(input("Enter a number"))
even = 0
odd = 0
for i in range(number):
        if i % 2 == 0:
                even = even + i
        else:
                odd = odd + i
print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")
```

Output

```
Enter a number 10
Sum of Even numbers are 20 and Odd numbers are 25
```

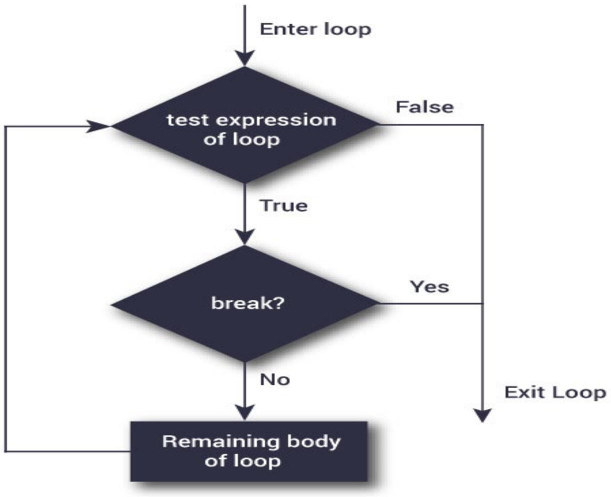
f)

```
# Write a Program to Find the Factorial of a Number
number = int(input('Enter a number'))
factorial = 1
if number < 0:
        print("Factorial doesn't exist for negative numbers")
elif number == 0:
        print('The factorial of 0 is 1')
else:
        for i in range(1, number + 1):
                factorial = factorial * i
        print(f"The factorial of number {number} is {factorial}")
```

- **<u>The ‘break’ and ‘continue’ statements</u>**

<u>break statement</u>

Terminates the loop statement and transfers execution to the statement immediately following the loop. With the ‘break’ statement we can stop the loop before it has looped through all the items:



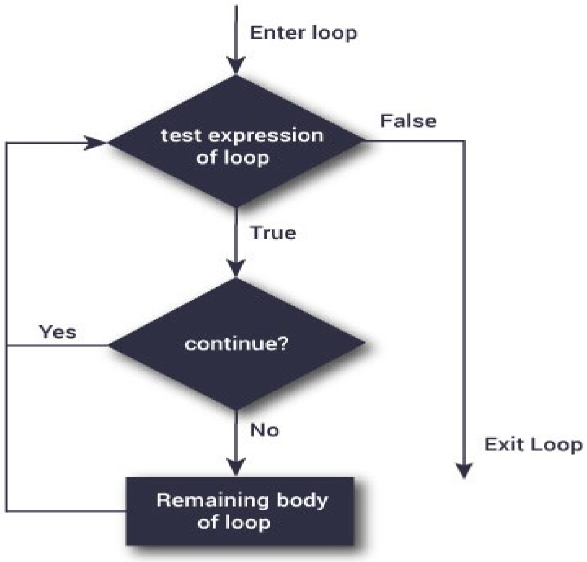
Examples:

a)

```
#Exit the loop when x is “banana”
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)                    # Prints “apple” only
```

continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. With the “continue” statement we can stop the current iteration of the loop, and continue with the next:



Example:

a)

```
# Do not print banana
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)                  # Prints “apple” & “cherry”
```

- **Catching Exceptions Using *try* and *except* statement in Python**

There are at least two distinguishable kinds of errors:

1. Syntax Errors

2. Exceptions

Syntax errors, also known as parsing errors, are perhaps the most common type of error in any programming language.

```
while True
        print("Hello World”)
```

Output

```
File "<ipython-input-3-c231969faf4f>", line 1
while True
^

SyntaxError: invalid syntax
```

The error is caused by a missing colon (':') after “True”.

<u>Exceptions</u>

Exception handling is one of the most important features of Python programming language that allows us to handle the errors caused by exceptions. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions.

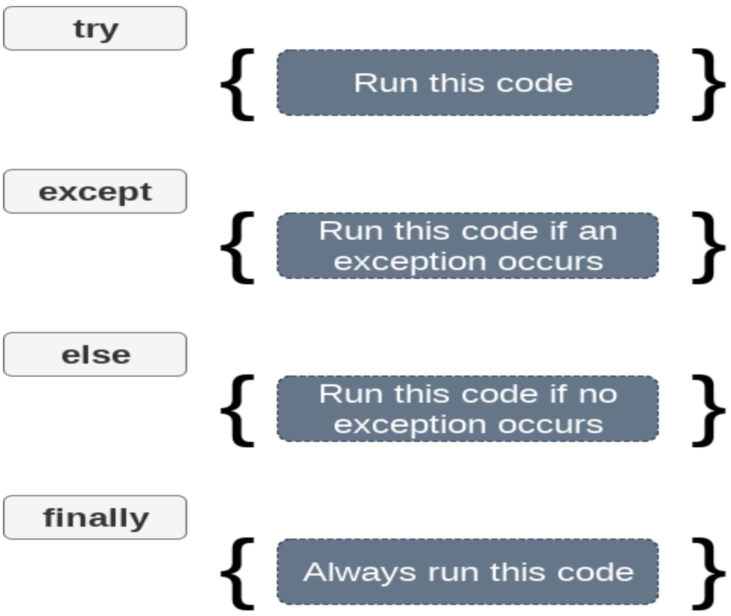
When the exceptions are not handled by programs it results in error messages as shown below.

```
1. >>> 10 * (1/0)
        Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
        ZeroDivisionError: division by zero

2. >>> 4 + var1*3                # Assume variable ‘var1’ is not defined before
        Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
        NameError: name 'var1' is not defined

3. >>> '2' + 2
        Traceback (most recent call last):
         File "<stdin>", line 1, in <module>
        TypeError: Can't convert 'int' object to str implicitly
```

Example 1 shows “ZeroDivisionError” exception
Example 2 shows “NameError” exception
Example 3 shows “TypeError” exception

- **Exception Handling Using *try…except…finally***

Note: Run-time errors are those errors that occur during the execution of the program. These errors are not detected by the Python interpreter, because the code is syntactically correct.


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code

Examples:

a)

```
while True:
    try:
        number = int(input("Please enter a number: "))
        print(f"The number you have entered is {number}")
        break
    except ValueError:
        print("Oops! That was no valid number. Try again…")
```

Output with incorrect and correct input

```
Please enter a number: abcd
Oops! That was no valid number. Try again…
Please enter a number: 4
The number you have entered is 4
```

b)

```
# Program to Check for ZeroDivisionError Exception
# but no check of ValueError exception
try:
    x = int(input("Enter value for x: "))
    y = int(input("Enter value for y: "))
    result = x / y
except ZeroDivisionError:
    print("Division by zero!")
else:
    print(f"Result is {result}")
finally:
    print("Executing finally clause")
```

Output:

Case 1:

```
Enter value for x: 6
Enter value for y: 2
Result is 3.0
Executing finally clause
```

```
Case 2:
Enter value for x: 5
Enter value for y: 0
Division by zero!
Executing finally clause
```

```
Case 3:              # Unhandled exception is raised
Enter value for x: q
Traceback (most recent call last):
  File "C:\Users\malaymitra\Desktop\test.py", line 1, in <module>
    x = int(input("Enter value for x: "))
ValueError: invalid literal for int() with base 10: 'q'
```

```
# Modified program to handle "ValueError" exception
try:
    x = int(input("Enter value for x: "))
    y = int(input("Enter value for y: "))
    result = x / y
except ZeroDivisionError:
    print("Division by zero!")
except ValueError:
    print('Error in wrong data type')
else:
    print(f"Result is {result}")
finally:
    print("Executing finally clause")
```

One can also mention a generic exception (mentioned below) statement to catch any unhandled exceptions which are not handled in the program.

```
except:
        print("Something else went wrong")
```

c)

```
# Even following code would generate an Exception if variable 'x' is not defined
# and one can trap the Exception message and print it
try:
        print(x)
except NameError as e:     # Trapping the message in an object called 'e'
        print(e)           # 'e' variable is an object of class 'NameError'
```

Output:

```
name 'x' is not defined
```