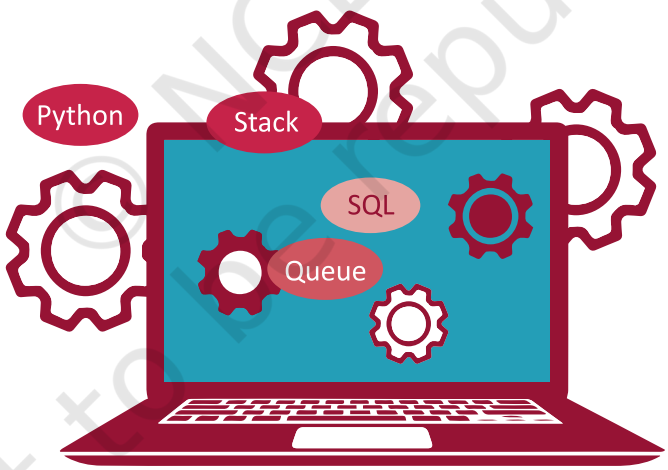
COMPUTER SCIENCE

TEXTBOOK FOR CLASS XII

12130



विद्यया ऽमृतमश्नुते
एन सी ई आर टी
NCERT

राष्ट्रीय शैक्षिक अनुसंधान और प्रशिक्षण परिषद्
NATIONAL COUNCIL OF EDUCATIONAL RESEARCH AND TRAINING

Reprint 2025-26

12130 – COMPUTER SCIENCE
Textbook for Class XII

ISBN 978-93-5292-338-0

First Edition
September 2020 Bhadrapada 1942

Reprinted
September 2021 Bhadrapada 1943
December 2021 Agrahayana 1943
October 2022 Kartika 1944
March 2024 Chaitra 1946

PD 25T SU

₹ 250.00

Printed on 80 GSM paper with NCERT watermark

Published at the Publication Division by the Secretary, National Council of Educational Research and Training, Sri Aurobindo Marg, New Delhi 110 016 and printed at Shree Vrindavan Graphics (P) Ltd., E-34, Sector-7, Noida 201 301 Uttar Pradesh

OFFICES OF THE PUBLICATION DIVISION, NCERT

NCERT Campus
Sri Aurobindo Marg
New Delhi 110 016 **Phone : 011-26562708**

108, 100 Feet Road
Hosdakere Halli Extension
Banashankari III Stage
Bengaluru 560 085 **Phone : 080-26725740**

Navjivan Trust Building
P.O.Navjivan
Ahmedabad 380 014 **Phone : 079-27541446**

CWC Campus
Opp. Dhankal Bus Stop
Panihati
Kolkata 700 114 **Phone : 033-25530454**

CWC Complex
Maligaon
Guwahati 781 021 **Phone : 0361-2674869**

Publication Team

Head, Publication Division	:	*Anup Kumar Rajput*
Chief Editor	:	*Shveta Uppal*
Chief Production Officer	:	*Arun Chitkara*
Chief Business Manager (In charge)	:	*Amitabh Kumar*
Assistant Production Officer	:	*Sunil Kumar*

Cover and Layout
Meetu Sharma, DTP Operator, DESM

Foreword

Computer science as a discipline has evolved over the years and has emerged as a driving force of our socio-economic activities. It has made continuous inroads into diverse areas — be it business, commerce, science, technology, sports, health, transportation or education. With the advent of computer and communication technologies, there has been a paradigm shift in teaching-learning at the school level. The role and relevance of this discipline is in focus because the expectations from the school pass-outs have grown to be able to meet the challenges of the 21st century. Today, we are living in an interconnected world where computer-based applications influence the way we learn, communicate, commute or even socialise!

There is a demand for software engineers in various fields like manufacturing, services, etc. Today, there are a large number of successful startups delivering different services through software applications. All these have resulted in generating interest for this subject among students as well as parents.

Development of logical thinking, reasoning and problem-solving skills are fundamental building blocks for knowledge acquisition at the higher level. Computer plays a key role in problem solving with focus on logical representation or reasoning and analysis.

This textbook focuses on the fundamental concepts and problem-solving skills while opening a window to the emerging and advanced areas of computer science. The newly developed syllabus has dealt with the dual challenge of reducing curricular load as well as introducing this ever evolving discipline. This textbook also provides space to Computational Thinking and Artificial Intelligence, which envisaged in National Education Policy, 2020.

As an organisation committed to systemic reforms and continuous improvement in the quality of its products, NCERT welcomes comments and suggestions which will enable us to revise the content of the textbook.

Hrushikesh Senapaty
Director
National Council of Educational
Research and Training

New Delhi
August 2020

Preface

In the present education system of our country, specialised or discipline based courses are introduced at the higher secondary stage. This stage is crucial as well as challenging because of the transition from general to discipline-based curriculum. The syllabus at this stage needs to have sufficient rigour and depth while remaining mindful of the comprehension level of the learners. Further, the textbook should not be heavily loaded with content.

Computers have permeated in every facet of life. Study of basic concepts of computer science has been desirable in education. There are courses offered in the name of Computer Science, Information and Communication Technology (ICT), Information Technology (IT), etc., by various boards and schools up to secondary stage, as optional. These mainly focus on using computer for word processing, presentation tools and application software.

Computer Science (CS) at the higher secondary stage of school education is also offered as an optional subject. At this stage, students usually opt for CS with an aim of pursuing a career in software development or related areas, after going through professional courses at higher levels. Therefore, at higher secondary stage, the curriculum of CS introduces basics of computing and sufficient conceptual background of Computer Science.

The primary focus is on fostering the development of computational thinking and problem-solving skills. This book has 13 chapters covering the following broader themes:

- Data Structure: understanding of important data structure Stack, Queue; Searching and Sorting techniques.
- Database: basic understanding of data, database concepts, and relational database management system using MySQL. Structured query language—data definition, data manipulation and data querying.
- Programming: handling errors and exceptions in programs written in Python; handling files and performing file operations in Python.
- Network and Communication: fundamentals of Computers networks, devices, topologies, Internet, Web and IoT, DNS. Basics of Data communication—transmission channel, media; basics of protocols, mobile communication generations.
- Security Aspects: introduction to basic concepts related to network and Internet security, threats and prevention.

Each chapter has two additional components—(i) activities and (ii) think and reflect for self assessment while learning as well as to generate further interest in the learner. A number of hands-on examples are given to gradually explain methodology to solve different types of problems across the Chapters. The programming examples as well as the exercises in the

chapters are required to be solved in a computer and verify with the given outputs.

Box items are pinned inside the chapters either to explain related concepts or to describe additional information related to the topic covered in that section. However, these box-items are not to be assessed through examinations.

Project Based Learning given as the end includes exemplar projects related to real-world problems. Teachers are supposed to assign these or similar projects to be developed in groups. Working in such projects may promote peer-learning, team spirit and responsiveness.

The chapters have been written by involving practicing teachers as well as subject experts. Several iterations have resulted into this book. Thanks are due to the authors and reviewers for their valuable contribution. I would like to place on record appreciation for Professor Om Vikas for leading the review activities of the book as well as for his guidance and motivation to the development team throughout. Comments and suggestions are welcome.

New Delhi
20 August 2020

Rejaul Karim Barbhuiya
Assistant Professor
Central Institute of
Educational Technology

Textbook Development Committee

Acknowledgements

The National Council of Educational Research and Training acknowledges the valuable contributions of the individuals and organisations involved in the development of Computer Science textbook for Class XII.

The Council expresses its gratitude to the syllabus development team including MPS Bhatia, *Professor*, Netaji Subhas Institute of Technology, Delhi; T. V. Vijay Kumar, *Professor*, School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi; Zahid Raza, *Associate Professor*, School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi; Vipul Shah, *Principal Scientist*, Tata Consultancy Services, and the CSpathshala team; Aasim Zafar, *Associate Professor*, Department of Computer Science, Aligarh Muslim University, Aligarh; Faisal Anwer, *Assistant Professor*, Department of Computer Science, Aligarh Muslim University, Aligarh; Smruti Ranjan Sarangi, *Associate Professor*, Department of Computer Science and Engineering, Indian Institute of Technology, Delhi; Vikram Goyal, *Associate Professor*, Indraprastha Institute of Information Technology (IIIT), Delhi; and Mamur Ali, *Assistant Professor*, Department of Teacher Training and Non-formal Education (IASE), Faculty of Education, Jamia Millia Islamia, New Delhi.

The Council is thankful to the following resource persons for providing valuable inputs in developing this book — Veer Sain Dixit, *Assistant Professor*, Atma Ram Sanatan Dharma College, University of Delhi; Mukesh Kumar, DPS RK Puram, Delhi; Aswin K. Dash, Mother's International School, Delhi; Anamika Gupta, *Assistant Professor*, Shaheed Sukhdev College of Business Studies, University of Delhi, Sajid Yousuf Bhat, *Assistant Professor*, University of Kashmir, Jammu and Kashmir.

The council is grateful to Prof. Sunita Farkya, *Head*, Department of Education in Science and Mathematics, NCERT and Prof. Amarendra P. Behera, *Joint Director*, CIET, NCERT for their valuable cooperation and support throughout the development of this book.

The Council also gracefully acknowledges the contributions of Meetu Sharma, *Graphic Designer cum DTP Operator*; Kanika Walecha, *DTP Operator*; Pooja, *Junior Project Fellow*; in shaping this book. The contributions of the office of the APC, DESM and Publication Division, NCERT, New Delhi, in bringing out this book are also duly acknowledged.

The Council also acknowledges the contribution of Ankeeta Bezboruah *Assistant Editor* (Contractual) Publication Division, NCERT for copy editing this book. The efforts of Naresh Kumar, *DTP Operator* (Contractual), Publication Division, NCERT are also acknowledged.

CONTENTS

THE CONSTITUTION OF INDIA

PREAMBLE

WE, THE PEOPLE OF INDIA, having solemnly resolved to constitute India into a [1]**[SOVEREIGN SOCIALIST SECULAR DEMOCRATIC REPUBLIC]** and to secure to all its citizens :

JUSTICE, social, economic and political;

LIBERTY of thought, expression, belief, faith and worship;

EQUALITY of status and of opportunity; and to promote among them all

FRATERNITY assuring the dignity of the individual and the [2][unity and integrity of the Nation];

IN OUR CONSTITUENT ASSEMBLY this twenty-sixth day of November, 1949 do **HEREBY ADOPT, ENACT AND GIVE TO OURSELVES THIS CONSTITUTION.**

---

1. Subs. by the Constitution (Forty-second Amendment) Act, 1976, Sec.2, for "Sovereign Democratic Republic" (w.e.f. 3.1.1977)
2. Subs. by the Constitution (Forty-second Amendment) Act, 1976, Sec.2, for "Unity of the Nation" (w.e.f. 3.1.1977)

Chapter 1

Exception Handling in Python

12130CH01

"I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimization. Clean code does one thing well."

— Bjarne Stroustrup

In this Chapter

1.1 Introduction

Sometimes while executing a Python program, the program does not execute at all or the program executes but generates unexpected output or behaves abnormally. These occur when there are syntax errors, runtime errors or logical errors in the code. In Python, exceptions are errors that get triggered automatically. However, exceptions can be forcefully triggered and handled through program code. In this chapter, we will learn about exception handling in Python programs.

1.2 Syntax Errors

Syntax errors are detected when we have not followed the rules of the particular programming language while writing a program. These errors are also known as *parsing errors*. On encountering a syntax error, the interpreter does not execute the program unless we rectify the errors, save and

rerun the program. When a syntax error is encountered while working in shell mode, Python displays the name of the error and a small description about the error as shown in Figure 1.1.

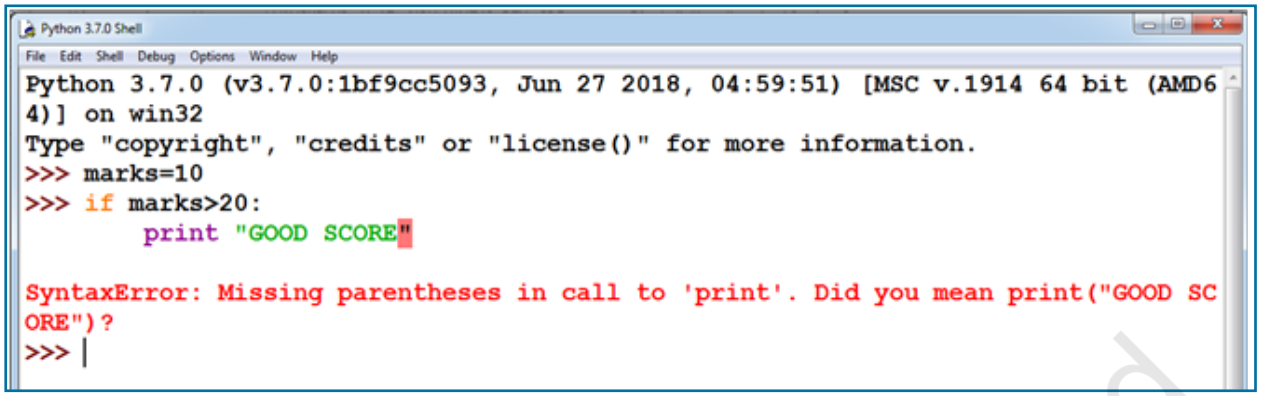


Figure 1.1: A syntax error displayed in Python shell mode

So, a syntax error is reported by the Python interpreter giving a brief explanation about the error and a suggestion to rectify it.

Similarly, when a syntax error is encountered while running a program in script mode as shown in Figure 1.2, a dialog box specifying the name of the error (Figure 1.3) and a small description about the error is displayed.

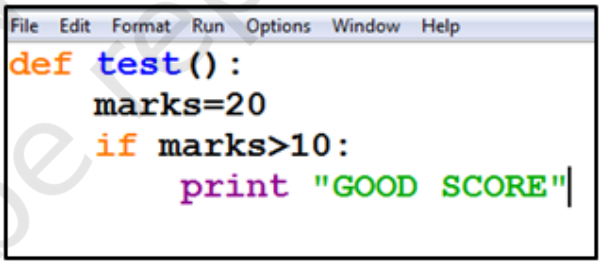


Figure 1.2: An error in the script

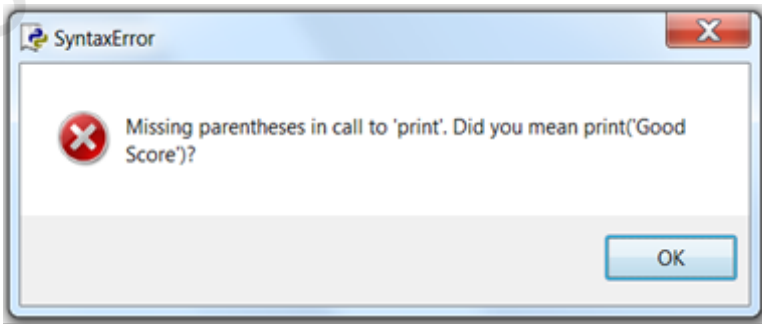


Figure 1.3: Error dialog box

1.3 Exceptions

Even if a statement or expression is syntactically correct, there might arise an error during its execution. For example, trying to open a file that does not exist, division by zero and so on. Such types of errors might disrupt the normal execution of the program and are called exceptions.

An exception is a Python object that represents an error. When an error occurs during the execution of a program, an exception is said to have been raised. Such an exception needs to be handled by the programmer so that the program does not terminate abnormally. Therefore, while designing a program, a programmer may anticipate such erroneous situations that may arise during its execution and can address them by including appropriate code to handle that exception.

It is to be noted that SyntaxError shown at Figures 1.1 and 1.3 is also an exception. But, all other exceptions are generated when a program is syntactically correct.

1.4 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 in Table 1.1.

Table 1.1 Built-in exceptions in Python

S. No	Name of the Built-in Exception	Explanation
1.	SyntaxError	It is raised when there is an error in the syntax of the Python code.
2.	ValueError	It is raised when a built-in method or operation receives an argument that has the right data type but mismatched or inappropriate values.
3.	IOError	It is raised when the file specified in a program statement cannot be opened.

Reprint 2025-26

4	KeyboardInterrupt	It is raised when the user accidentally hits the Delete or Esc key while executing a program due to which the normal flow of the program is interrupted.
5	ImportError	It is raised when the requested module definition is not found.
6	EOFError	It is raised when the end of file condition is reached without reading any data by input().
7	ZeroDivisionError	It is raised when the denominator in a division operation is zero.
8	IndexError	It is raised when the index or subscript in a sequence is out of range.
9	NameError	It is raised when a local or global variable name is not defined.
10	IndentationError	It is raised due to incorrect indentation in the program code.
11	TypeError	It is raised when an operator is supplied with a value of incorrect data type.
12	OverFlowError	It is raised when the result of a calculation exceeds the maximum limit for numeric data type.

Figure 1.4 shows the built-in exceptions viz, ZeroDivisionError, NameEError, and TypeError raised by the Python interpreter in different situations.

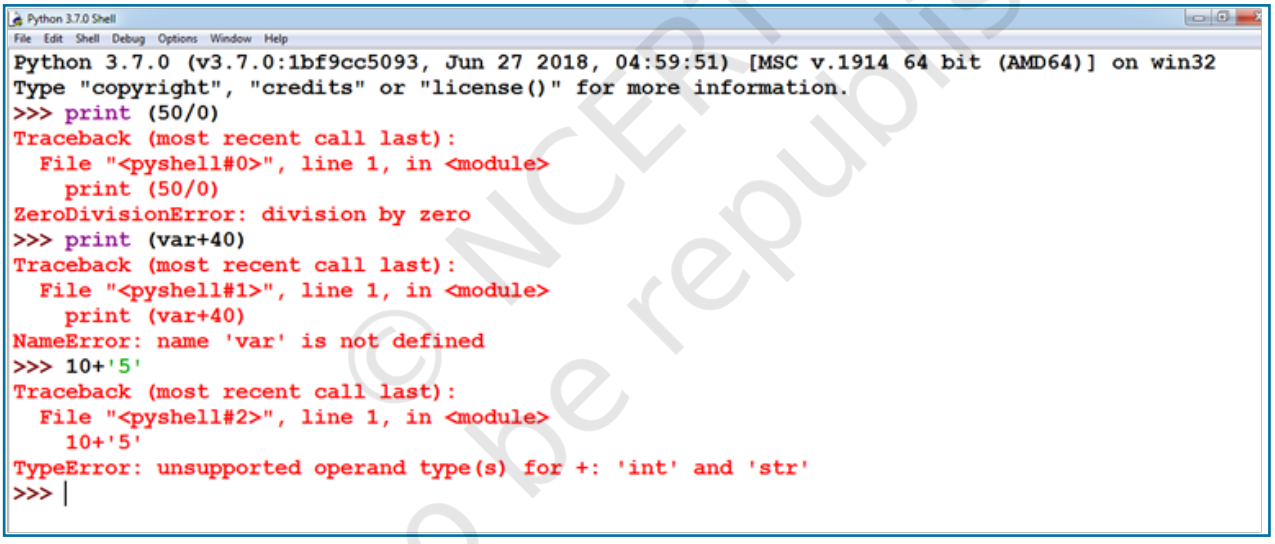


Figure 1.4: Example of built-in exceptions

A programmer can also create custom exceptions to suit one's requirements. These are called user-defined exceptions. We will learn how to handle exceptions in the next section.

1.5 Raising Exceptions

Each time an error is detected in a program, the Python interpreter raises (throws) an exception. Exception

handlers are designed to execute when a specific exception is raised. Programmers can also forcefully raise exceptions in a program using the raise and assert statements. Once an exception is raised, no further statement in the current block of code is executed. So, raising an exception involves interrupting the normal flow execution of program and jumping to that part of the program (exception handler code) which is written to handle such exceptional situations.

1.5.1 The `raise` Statement

The raise statement can be used to throw an exception. The syntax of raise statement is:

```
raise exception-name[(optional argument)]
```

The argument is generally a string that is displayed when the exception is raised. For example, when an exception is raised as shown in Figure 1.5, the message "OOPS : An Exception has occurred" is displayed along with a brief description of the error.

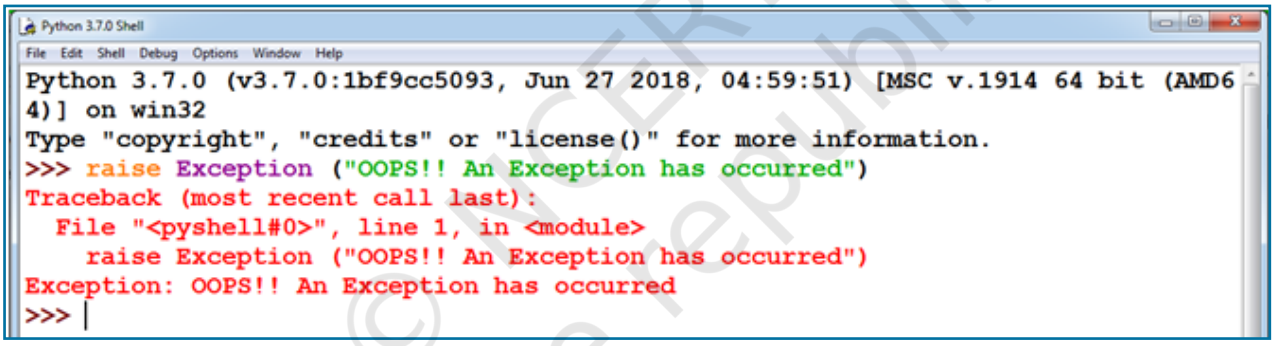


Figure 1.5: Use of the raise statement to throw an exception

The error detected may be a built-in exception or may be a user-defined one. Consider the example given in Figure 1.6 that uses the `raise` statement to raise a built-in exception called `IndexError`.

Note: In this case, the user has only raised the exception but has not displayed any error message explicitly.

In Figure 1.6, since the value of variable length is greater than the length of the list *numbers*, an `IndexError` exception will be raised. The statement following the raise statement will not be executed. So the message "NO EXECUTION" will not be displayed in this case.

Reprint 2025-26

As we can see in Figure 1.6, in addition to the error message displayed, Python also displays a stack Traceback. This is a structured block of text that contains information about the sequence of function calls that have been made in the branch of execution of code in which the exception was raised. In Figure 1.6, the error has been encountered in the most recently called function that has been executed.

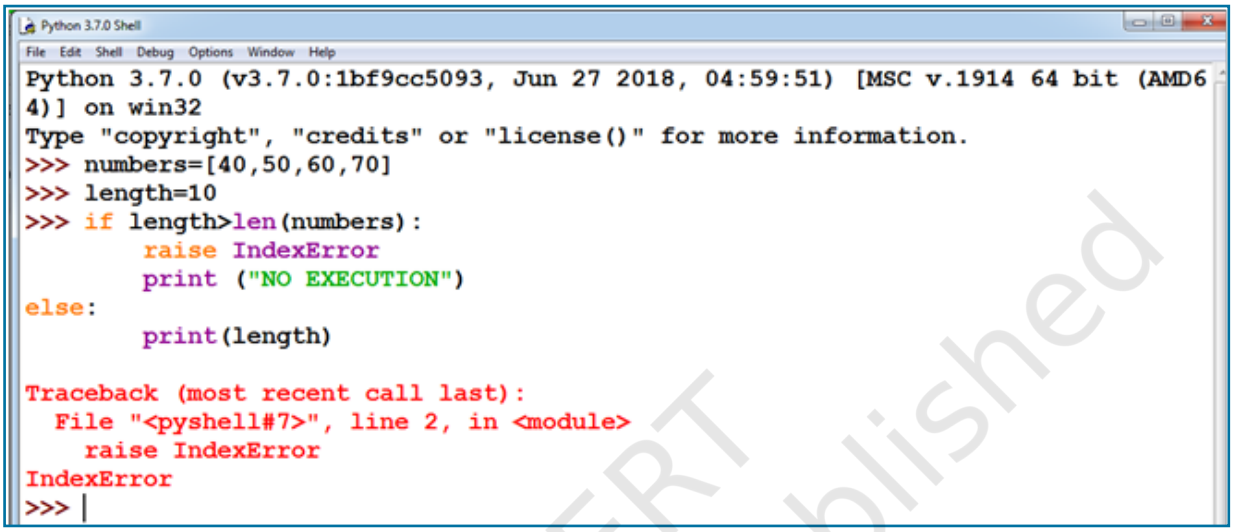


Figure 1.6: Use of raise statement with built-in exception

Note: We will learn about Stack in Chapter 3.

1.5.2 The `assert` Statement

An assert statement in Python is used to test an expression in the program code. If the result after testing comes false, then the exception is raised. This statement is generally used in the beginning of the function or after a function call to check for valid input. The syntax for assert statement is:

```
assert Expression[,arguments]
```

On encountering an assert statement, Python evaluates the expression given immediately after the assert keyword. If this expression is false, an AssertionError exception is raised which can be handled like any other exception. Consider the code given in Program 1-1.

Program 1-1 Use of assert statement

```
print("use of assert statement")
def negativecheck(number):
    assert(number>=0), "OOPS... Negative Number"
```

```
        print(number*number)
    print (negativecheck(100))
    print (negativecheck(-350))
```

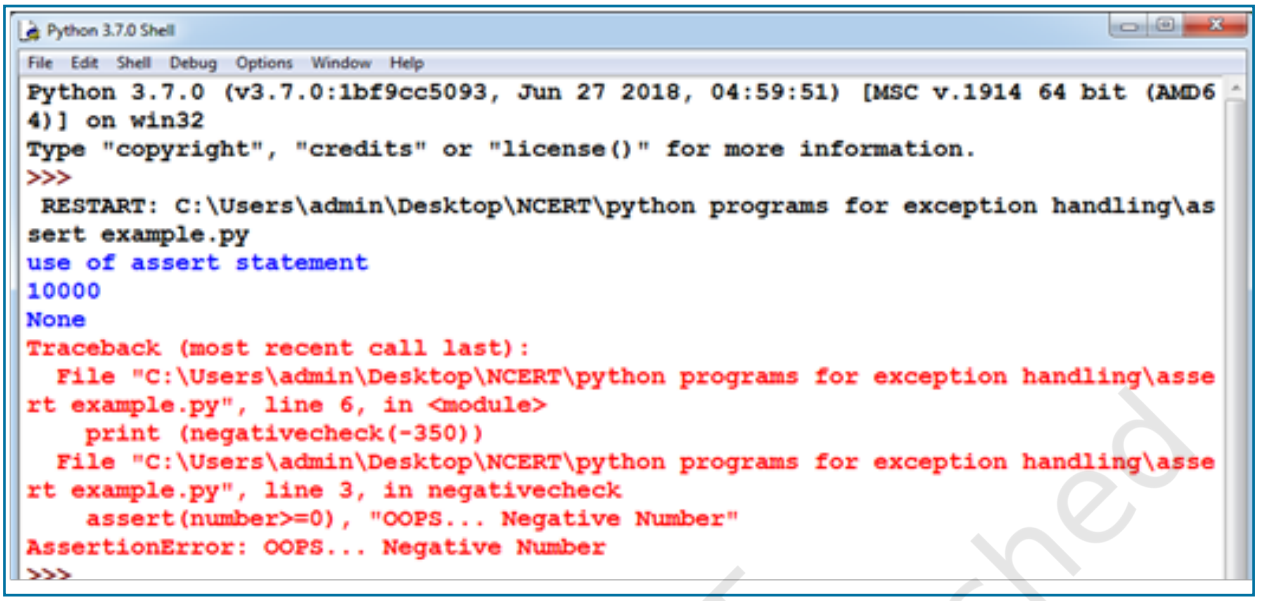


Figure 1.7: Output of Program 1-1.

In the code, the assert statement checks for the value of the variable number. In case the number gets a negative value, `AssertionError` will be thrown, and subsequent statements will not be executed. Hence, on passing a negative value (-350) as an argument, it results in `AssertionError` and displays the message "OOPS.... Negative Number". The output of the code is shown in Figure 1.7.

1.6 Handling Exceptions

Each and every exception has to be handled by the programmer to avoid the program from crashing abruptly. This is done by writing additional code in a program to give proper messages or instructions to the user on encountering an exception. This process is known as *exception handling*.

1.6.1 Need for Exception Handling

Exception handling is being used not only in Python programming but in most programming languages like C++, Java, Ruby, etc. It is a useful technique that helps in capturing runtime errors and handling them so as to avoid the program getting crashed. Following are some

Reprint 2025-26

of the important points regarding exceptions and their handling:

- Python categorises exceptions into distinct types so that specific exception handlers (code to handle that particular exception) can be created for each type.
- Exception handlers separate the main logic of the program from the error detection and correction code. The segment of code where there is any possibility of error or exception, is placed inside one block. The code to be executed in case the exception has occurred, is placed inside another block. These statements for detection and reporting the exception do not affect the main logic of the program.
- The compiler or interpreter keeps track of the exact position where the error has occurred.
- Exception handling can be done for both user-defined and built-in exceptions.

1.6.2 Process of Handling Exception

When an error occurs, Python interpreter creates an object called the *exception object*. This object contains information about the error like its type, file name and position in the program where the error has occurred. The object is handed over to the runtime system so that it can find an appropriate code to handle this particular exception. This process of creating an exception object and handing it over to the runtime system is called *throwing* an exception. It is important to note that when an exception occurs while executing a particular program statement, the control jumps to an exception handler, abandoning execution of the remaining program statements.

A runtime system refers to the execution of the statements given in the program. It is a complex mechanism consisting of hardware and software that comes into action as soon as the program, written in any programming language, is put for execution.

The runtime system searches the entire program for a block of code, called the *exception handler* that can handle the raised exception. It first searches for the method in which the error has occurred and the exception has been raised. If not found, then it searches the method from which this method (in which exception was raised) was called. This hierarchical search in reverse order continues till the exception handler is found. This entire list of methods is known as *call stack*. When a suitable handler is found in the call stack, it is executed by the runtime process. This process of

executing a suitable handler is known as *catching the exception*. If the runtime system is not able to find an appropriate exception after searching all the methods in the call stack, then the program execution stops.

The flowchart in Figure 1.8 describes the exception handling process.

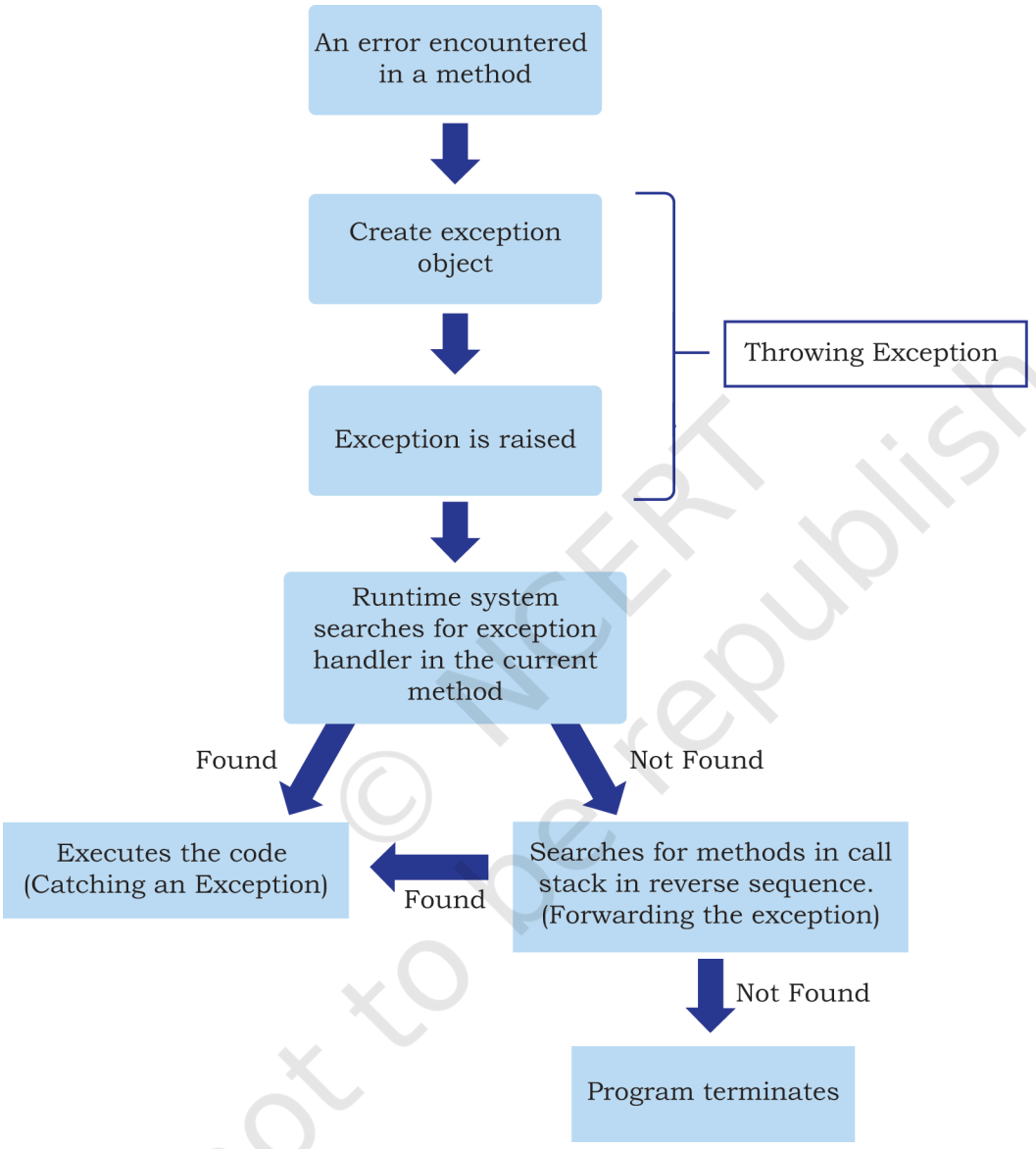


Figure 1.8: Steps of handling exception

1.6.3 Catching Exceptions

An exception is said to be caught when a code that is designed to handle a particular exception is executed. Exceptions, if any, are caught in the `try` block and

handled in the except block. While writing or debugging a program, a user might doubt an exception to occur in a particular part of the code. Such suspicious lines of codes are put inside a try block. Every try block is followed by an except block. The appropriate code to handle each of the possible exceptions (in the code inside the try block) are written inside the except clause.

While executing the program, if an exception is encountered, further execution of the code inside the try block is stopped and the control is transferred to the except block. The syntax of try … except clause is as follows:

```
try:
   [ program statements where exceptions might occur]
except [exception-name]:
   [ code for exception handling if the exception-name error is
   encountered]
```

Consider the Program 1-2 given below:

Program 1-2 Using try..except block

```
print ("Practicing for try block")
try:
    numerator=50
    denom=int(input("Enter the denominator"))
    quotient=(numerator/denom)
    print(quotient)
    print ("Division performed successfully")
except ZeroDivisionError:
    print ("Denominator as ZERO.... not allowed")
print("OUTSIDE try..except block")
```

In Program 1-2, the ZeroDivisionError exception is handled. If the user enters any non-zero value as denominator, the quotient will be displayed along with the message "Division performed successfully", as shown in Figure 1.10. The except clause will be skipped in this case. So, the next statement after the try..except block is executed and the message "OUTSIDE try.. except block" is displayed.

However, if the user enters the value of denom as zero (0), then the execution of the try block will stop. The control will shift to the except block and the message "Denominator as Zero…. not allowed" will be displayed, as shown in Figure 1.11. Thereafter, the

Reprint 2025-26

statement following the try..except block is executed and the message "`OUTSIDE try..except block`" is displayed in this case also.

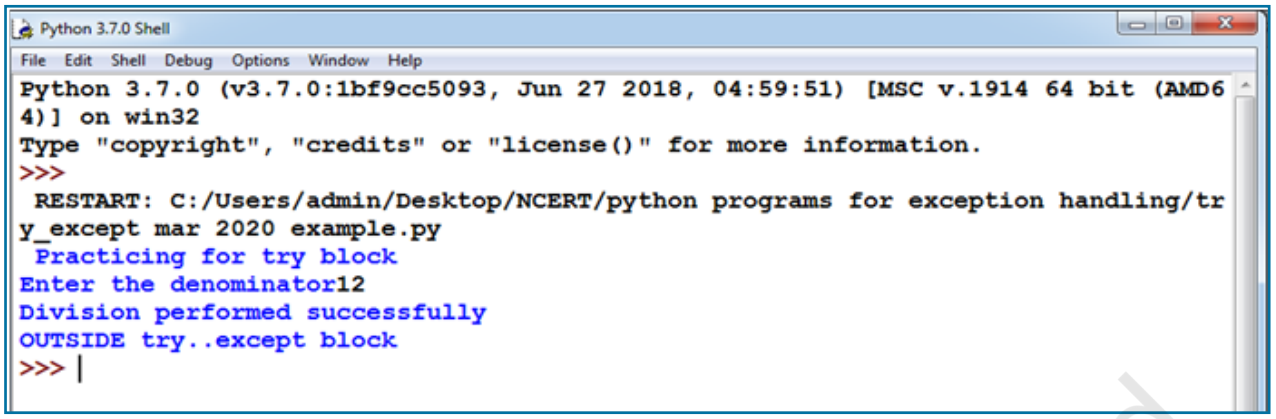


Figure 1.9: Output without an error

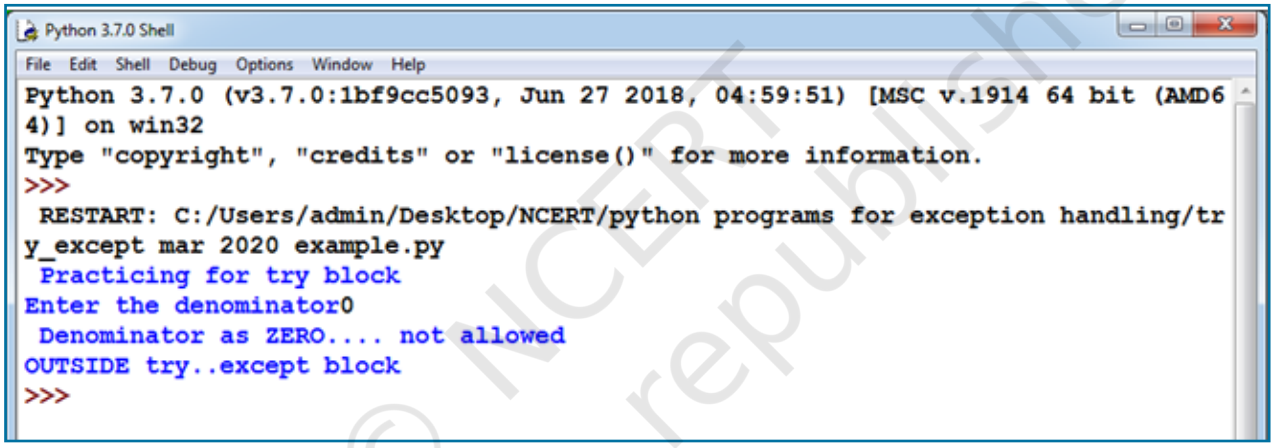


Figure 1.10: Output with exception raised

Sometimes, a single piece of code might be suspected to have more than one type of error. For handling such situations, we can have multiple except blocks for a single try block as shown in the Program 1-3.

Program 1-3 Use of multiple except clauses

```
print ("Handling multiple exceptions")
try:
    numerator=50
    denom=int(input("Enter the denominator: "))
    print (numerator/denom)
    print ("Division performed successfully")
except ZeroDivisionError:
    print ("Denominator as ZERO is not allowed")
except ValueError:
    print ("Only INTEGERS should be entered")
```

In the code, two types of exceptions (ZeroDivisionError and `ValueError`) are handled using two `except` blocks for a single `try` block. When an exception is raised, a search for the matching `except` block is made till it is handled. If no match is found, then the program terminates.

However, if an exception is raised for which no handler is created by the programmer, then such an exception can be handled by adding an except clause without specifying any exception. This except clause should be added as the last clause of the try..except block. The Program 1-4 given below along with the output given in Figure 1.11 explains this.

Program 1-4 Use of except without specifying an exception

```
print ("Handling exceptions without naming them")
try:
    numerator=50
    denom=int(input("Enter the denominator"))
    quotient=(numerator/denom)
    print ("Division performed successfully")
except ValueError:
    print ("Only INTEGERS should be entered")
except:
    print(" OOPS.....SOME EXCEPTION RAISED")
```

If the above code is executed, and the denominator entered is 0 (zero) , the handler for `ZeroDivisionError` exception will be searched. Since it is not present, the last except clause (without any specified exception) will be executed , so the message " OOPS.....SOME EXCEPTION RAISED" will be displayed.

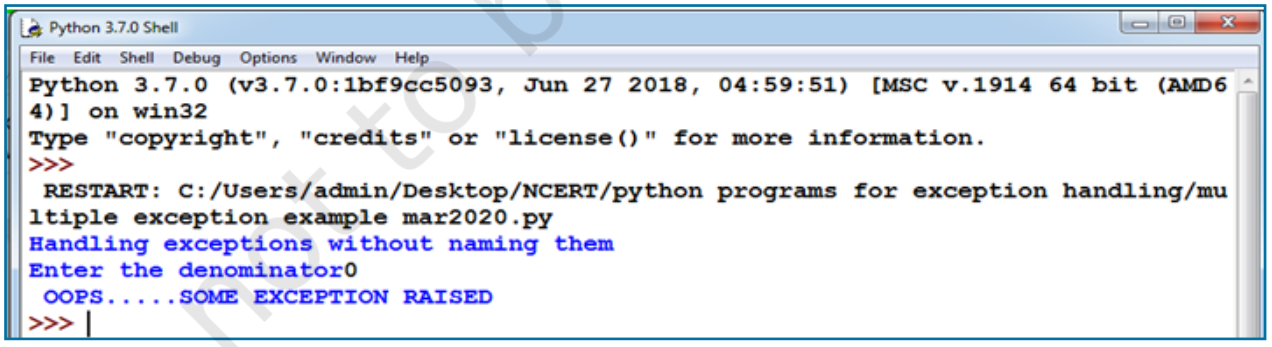


Figure 1.11: Output of Program 1-4

1.6.4 `try...except...else` clause

We can put an optional `else` clause along with the `try...except` clause. An `except` block will be executed

Reprint 2025-26

only if some exception is raised in the `try` block. But if there is no error then none of the `except` blocks will be executed. In this case, the statements inside the `else` clause will be executed. Program 1-5 along with its output explains the use of else block with the try... except block.

Program 1-5 Use of else clause

```
print ("Handling exception using try...except...else")
try:
    numerator=50
    denom=int(input("Enter the denominator: "))
    quotient=(numerator/denom)
    print ("Division performed successfully")

except ZeroDivisionError:
    print ("Denominator as ZERO is not allowed")

except ValueError:
    print ("Only INTEGERS should be entered")

else:
   print ("The result of division operation is ", quotient)
```

Output:

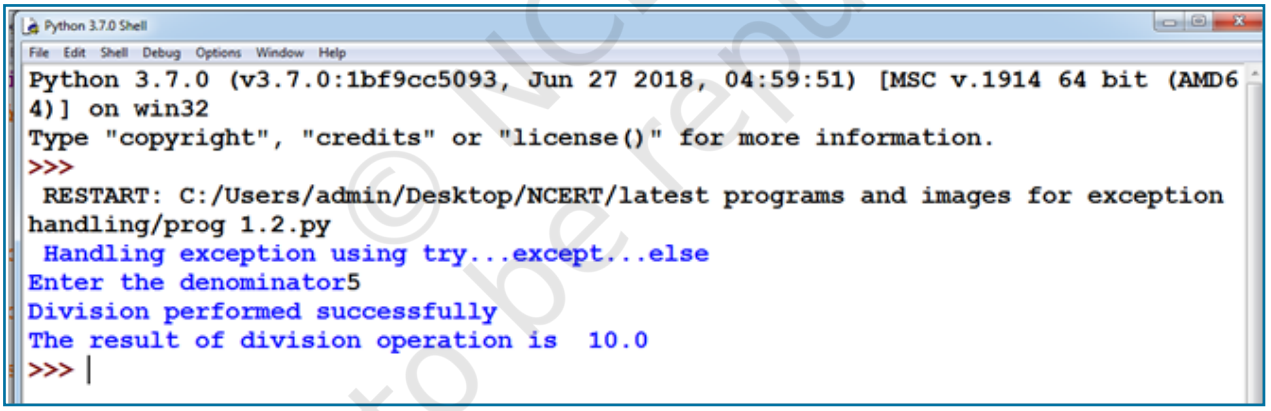


Figure 1.12: Output of Program 1-5.

1.7 Finally Clause

The try statement in Python can also have an optional finally clause. The statements inside the finally block are always executed regardless of whether an exception has occurred in the try block or not. It is a common practice to use `finally` clause while working with files to ensure that the file object is closed. If used, `finally` should always be placed at the end of `try` clause, after all `except` blocks and the `else` block.

Program 1-6 Use of finally clause

```
print ("Handling exception using try...except...else...finally")
try:
  numerator=50
  denom=int(input("Enter the denominator: "))
  quotient=(numerator/denom)
  print ("Division performed successfully")
except ZeroDivisionError:
  print ("Denominator as ZERO is not allowed")
except ValueError:
  print ("Only INTEGERS should be entered")
else:
  print ("The result of division operation is ", quotient)
finally:
  print ("OVER AND OUT")
```

In the above program, the message "OVER AND OUT" will be displayed irrespective of whether an exception is raised or not.

1.6.1 Recovering and continuing with finally clause

If an error has been detected in the `try` block and the exception has been thrown, the appropriate except block will be executed to handle the error. But if the exception is not handled by any of the `except` clauses, then it is re-raised after the execution of the `finally` block. For example, Program 1.4 contains only the `except` block for `ZeroDivisionError`. If any other type of error occurs for which there is no handler code (except clause) defined, then also the `finally` clause will be executed first. Consider the code given in Program 1-7 to understand these concepts.

Program 1-7 Recovering through finally clause

```
print (" Practicing for try block")
try:
  numerator=50
  denom=int(input("Enter the denominator"))
  quotient=(numerator/denom)
  print ("Division performed successfully")
except ZeroDivisionError:
  print ("Denominator as ZERO is not allowed")
else:
  print ("The result of division operation is ", quotient)
finally:
  print ("OVER AND OUT")
```

While executing the above code, if we enter a non-numeric data as input, the `finally` block will be executed. So, the message "OVER AND OUT" will be displayed. Thereafter the exception for which handler is not present will be re-raised. The output of Program 1-7 is shown in Figure 1.13.

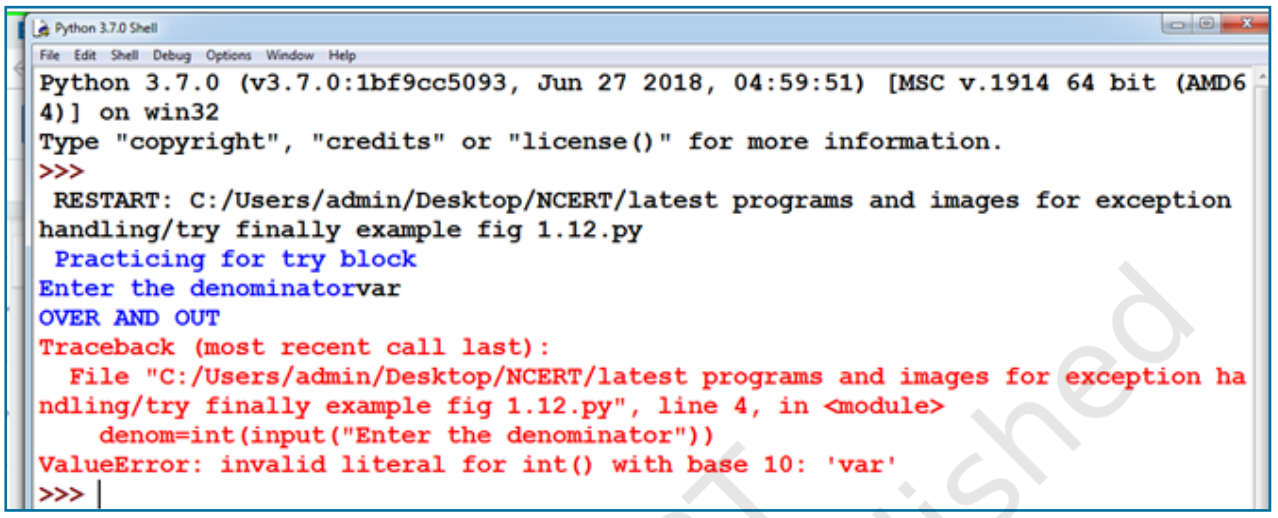


Figure 1.13: Output of Program 1-7

After execution of `finally` block, Python transfers the control to a previously entered try or to the next higher level default exception handler. In such a case, the statements following the `finally` block is executed. That is, unlike except, execution of the finally clause does not terminate the exception. Rather, the exception continues to be raised after execution of finally.

To summarise, we put a piece of code where there are possibilities of errors or exceptions to occur inside a try block. Inside each except clause we define handler codes to handle the matching exception raised in the try block. The optional else clause contains codes to be executed if no exception occurs. The optional finally block contains codes to be executed irrespective of whether an exception occurs or not.

SUMMARY

- Syntax errors or parsing errors are detected when we have not followed the rules of the particular programming language while writing a program.

Reprint 2025-26

- When syntax error is encountered, Python displays the name of the error and a small description about the error.
- The execution of the program will start only after the syntax error is rectified.
- An exception is a Python object that represents an error.
- Syntax errors are also handled as exceptions.
- The exception needs to be handled by the programmer so that the program does not terminate abruptly.
- When an exception occurs during execution of a program and there is a built-in exception defined for that, the error message written in that exception is displayed. The programmer then has to take appropriate action and handle it.
- Some of the commonly occurring built-in exceptions are `SyntaxError`, `ValueError`, `IOError`, `KeyboardInterrupt`, `ImportError`, `EOFError`, `ZeroDivisionError`, `IndexError`, `NameError`, `IndentationError`, `TypeError`, and `OverFlowerror`.
- When an error is encountered in a program, Python interpreter raises or throws an exception.
- Exception Handlers are the codes that are designed to execute when a specific exception is raised.
- Raising an exception involves interrupting the normal flow of the program execution and jumping to the exception handler.
- Raise and assert statements are used to raise exceptions.
- The process of exception handling involves writing additional code to give proper messages or instructions to the user. This prevents the program from crashing abruptly. The additional code is known as an exception handler.
- An exception is said to be caught when a code that is designed to handle a particular exception is executed.
- An exception is caught in the try block and handles in except block.

- The statements inside the finally block are always executed regardless of whether an exception occurred in the try block or not.

EXERCISE

1. "Every syntax error is an exception but every exception cannot be a syntax error." Justify the statement.
2. When are the following built-in exceptions raised? Give examples to support your answers.
   a) ImportError
   b) IOError
   c) NameError
   d) ZeroDivisionError
3. What is the use of a raise statement? Write a code to accept two numbers and display the quotient. Appropriate exception should be raised if the user enters the second number (denominator) as zero (0).
4. Use assert statement in Question No. 3 to test the division expression in the program.
5. Define the following:
   a) Exception Handling
   b) Throwing an exception
   c) Catching an exception
6. Explain catching exceptions using try and except block.
7. Consider the code given below and fill in the blanks.

```
print (" Learning Exceptions...")
try:
    num1= int(input ("Enter the first number"))
    num2=int(input("Enter the second number"))
    quotient=(num1/num2)
    print ("Both the numbers entered were correct")
except _____________:              # to enter only integers
    print (" Please enter only numbers")
except _____________:       # Denominator should not be zero
  print(" Number 2 should not be zero")
else:
  print(" Great .. you are a good programmer")
_____________:                  # to be executed at the end
  print(" JOB OVER... GO GET SOME REST")
```


Reprint 2025-26

8. You have learnt how to use math module in Class XI. Write a code where you use the wrong number of arguments for a method (say sqrt() or pow()). Use the exception handling process to catch the ValueError exception.
9. What is the use of finally clause? Use finally clause in the problem given in Question No. 7.

Reprint 2025-26

Chapter 2

File Handling in Python

12130CH02

There are many ways of trying to understand programs. People often rely too much on one way, which is called "debugging" and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.

— Robin Milner

In this Chapter

2.1 Introduction to Files

We have so far created programs in Python that accept the input, manipulate it and display the output. But that output is available only during execution of the program and input is to be entered through the keyboard. This is because the variables used in a program have a lifetime that lasts till the time the program is under execution. What if we want to store the data that were input as well as the generated output permanently so that we can reuse it later? Usually, organisations would want to permanently store information about employees, inventory, sales, etc. to avoid repetitive tasks of entering the same data. Hence, data are stored permanently on secondary storage devices for reusability. We store Python programs written in script mode with a .py extension. Each program is stored on the secondary device as a file. Likewise, the data entered, and the output can be stored permanently into a file.

Text files contain only the ASCII equivalent of the contents of the file whereas a .docx file contains many additional information like the author's name, page settings, font type and size, date of creation and modification, etc.

So, what is a file? A file is a named location on a secondary storage media where data are permanently stored for later access.

2.2. Types of Files

Computers store every file as a collection of 0s and 1s i.e., in binary form. Therefore, every file is basically just a series of bytes stored one after the other. There are mainly two types of data files — text file and binary file. A text file consists of human readable characters, which can be opened by any text editor. On the other hand, binary files are made up of non-human readable characters and symbols, which require specific programs to access its contents.

2.2.1 Text file

A text file can be understood as a sequence of characters consisting of alphabets, numbers and other special symbols. Files with extensions like .txt, .py, .csv, etc. are some examples of text files. When we open a text file using a text editor (e.g., Notepad), we see several lines of text. However, the file contents are not stored in such a way internally. Rather, they are stored in sequence of bytes consisting of 0s and 1s. In ASCII, UNICODE or any other encoding scheme, the value of each character of the text file is stored as bytes. So, while opening a text file, the text editor translates each ASCII value and shows us the equivalent character that is readable by the human being. For example, the ASCII value 65 (binary equivalent 1000001) will be displayed by a text editor as the letter 'A' since the number 65 in ASCII character set represents 'A'.

Each line of a text file is terminated by a special character, called the End of Line (EOL). For example, the default EOL character in Python is the newline (\n). However, other characters can be used to indicate EOL. When a text editor or a program interpreter encounters the ASCII equivalent of the EOL character, it displays the remaining file contents starting from a new line. Contents in a text file are usually separated by whitespace, but comma (,) and tab (\t) are also commonly used to separate values in a text file.

Activity 2.1

Create a text file using notepad and write your name and save it. Now, create a .docx file using Microsoft Word and write your name and save it as well. Check and compare the file size of both the files. You will find that the size of .txt file is in bytes whereas that of .docx is in KBs.

2.2.2 Binary Files

Binary files are also stored in terms of bytes (0s and 1s), but unlike text files, these bytes do not represent the ASCII values of characters. Rather, they represent the actual content such as image, audio, video, compressed versions of other files, executable files, etc. These files are not human readable. Thus, trying to open a binary file using a text editor will show some garbage values. We need specific software to read or write the contents of a binary file.

Binary files are stored in a computer in a sequence of bytes. Even a single bit change can corrupt the file and make it unreadable to the supporting application. Also, it is difficult to remove any error which may occur in the binary file as the stored contents are not human readable. We can read and write both text and binary files through Python programs.

2.3 Opening and Closing a Text File

In real world applications, computer programs deal with data coming from different sources like databases, CSV files, HTML, XML, JSON, etc. We broadly access files either to write or read data from it. But operations on files include creating and opening a file, writing data in a file, traversing a file, reading data from a file and so on. Python has the io module that contains different functions for handling files.

2.3.1 Opening a file

To open a file in Python, we use the open() function. The syntax of open() is as follows:

```
file_object= open(file_name, access_mode)
```

This function returns a file object called file handle which is stored in the variable `file_object`. We can use this variable to transfer data to and from the file (read and write) by calling the functions defined in the Python's `io module`. If the file does not exist, the above statement creates a new empty file and assigns it the name we specify in the statement.

The `file_object` has certain attributes that tells us basic information about the file, such as:

- `<file.closed>` returns true if the file is closed and false otherwise.

The `file_object` establishes a link between the program and the data file stored in the permanent storage.

Reprint 2025-26

- `<file.mode>` returns the access mode in which the file was opened.
- `<file.name>` returns the name of the file.

Activity 2.2

Some of the other file access modes are <rb+>, <wb>, <w+>, <ab>, <ab+>. Find out for what purpose each of these are used. Also, find the file offset positions in each case.

The `file_name` should be the name of the file that has to be opened. If the file is not in the current working directory, then we need to specify the complete path of the file along with its name.

The `access_mode` is an optional argument that represents the mode in which the file has to be accessed by the program. It is also referred to as processing mode. Here mode means the operation for which the file has to be opened like <r> for reading, <w> for writing, <+> for both reading and writing, <a> for appending at the end of an existing file. The default is the read mode. In addition, we can specify whether the file will be handled as binary (<b>) or text mode. By default, files are opened in text mode that means strings can be read or written. Files containing non-textual data are opened in binary mode that means read/write are performed in terms of bytes. Table 2.1 lists various file access modes that can be used with the open() method. The file offset position in the table refers to the position of the file object when the file is opened in a particular mode.

Table 2.1 File Open Modes

File Mode	Description	File Offset position
<r>	Opens the file in read-only mode.	Beginning of the file
<rb>	Opens the file in binary and read-only mode.	Beginning of the file
<r+> or <+r>	Opens the file in both read and write mode.	Beginning of the file
<w>	Opens the file in write mode. If the file already exists, all the contents will be overwritten. If the file doesn't exist, then a new file will be created.	Beginning of the file
<wb+> or <+wb>	Opens the file in read,write and binary mode. If the file already exists, the contents will be overwritten. If the file doesn't exist, then a new file will be created.	Beginning of the file
<a>	Opens the file in append mode. If the file doesn't exist, then a new file will be created.	End of the file
<a+> or <+a>	Opens the file in append and read mode. If the file doesn't exist, then it will create a new file.	End of the file

Consider the following example.

```
myObject=open("myfile.txt", "a+")
```


Reprint 2025-26

In the above statement, the file *myfile.txt* is opened in append and read modes. The file object will be at the end of the file. That means we can write data at the end of the file and at the same time we can also read data from the file using the file object named *myObject*.

2.3.2 Closing a file

Once we are done with the read/write operations on a file, it is a good practice to close the file. Python provides a `close()` method to do so. While closing a file, the system frees the memory allocated to it. The syntax of `close()` is:

```
file_object.close()
```

Here, file_object is the object that was returned while opening the file.

Python makes sure that any unwritten or unsaved data is flushed off (written) to the file before it is closed. Hence, it is always advised to close the file once our work is done. Also, if the file object is re-assigned to some other file, the previous file is automatically closed.

2.3.3 Opening a file using with clause

In Python, we can also open a file using with clause. The syntax of with clause is:

```
with open (file_name, access_mode) as file_
object:
```

The advantage of using with clause is that any file that is opened using this clause is closed automatically, once the control comes outside the with clause. In case the user forgets to close the file explicitly or if an exception occurs, the file is closed automatically. Also, it provides a simpler syntax.

```
with open("myfile.txt","r+") as myObject:
    content = myObject.read()
```

Here, we don't have to close the file explicitly using close() statement. Python will automatically close the file.

2.4 Writing to a Text File

For writing to a file, we first need to open it in write or append mode. If we open an existing file in write mode, the previous data will be erased, and the file object will be positioned at the beginning of the file. On the other

Reprint 2025-26

Think and Reflect

For a newly created file, is there any difference between write() and append() methods?

hand, in append mode, new data will be added at the end of the previous data as the file object is at the end of the file. After opening the file, we can use the following methods to write data in the file.

- write() - for writing a single string
- writelines() - for writing a sequence of strings

2.4.1 The write() method

write() method takes a string as an argument and writes it to the text file. It returns the number of characters being written on single execution of the write() method. Also, we need to add a newline character (\n) at the end of every sentence to mark the end of line.

Consider the following piece of code:

```
>>> myobject=open("myfile.txt",'w')
>>> myobject.write("Hey I have started
    #using files in Python\n")
41
>>> myobject.close()
```

On execution, write() returns the number of characters written on to the file. Hence, 41, which is the length of the string passed as an argument, is displayed.

Note: '\n' is treated as a single character

If numeric data are to be written to a text file, the data need to be converted into string before writing to the file. For example:

```
>>>myobject=open("myfile.txt",'w')
>>> marks=58
#number 58 is converted to a string using
#str()
>>> myobject.write(str(marks))
2
>>>myobject.close()
```

We can also use the flush() method to clear the buffer and write contents in buffer to the file. This is how programmers can forcefully write to the file as and when required.

The `write()` actually writes data onto a buffer. When the close() method is executed, the contents from this buffer are moved to the file located on the permanent storage.

2.4.2 The writelines() method

This method is used to write multiple strings to a file. We need to pass an iterable object like lists, tuple, etc. containing strings to the `writelines()` method. Unlike

Reprint 2025-26

write(), the writelines() method does not return the number of characters written in the file. The following code explains the use of writelines().

```
>>> myobject=open("myfile.txt",'w')
>>> lines = ["Hello everyone\n", "Writing
    #multiline strings\n", "This is the
    #third line"]
>>> myobject.writelines(lines)
>>>myobject.close()
```

On opening myfile.txt, using notepad, its content will appear as shown in Figure 2.1.

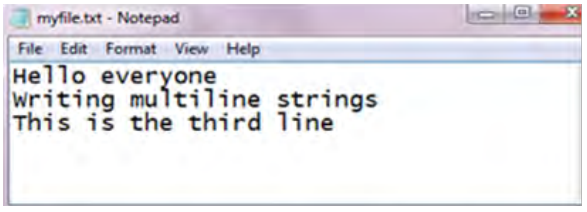


Figure 2.1: Contents of myfile.txt

Activity 2.3

Run the above code by replacing writelines() with write() and see what happens.

Think and Reflect

Can we pass a tuple of numbers as an argument to writelines()? Will it be written to the file or an error will be generated?

2.5 Reading from a Text File

We can write a program to read the contents of a file. Before reading a file, we must make sure that the file is opened in "r", "r+", "w+" or "a+" mode. There are three ways to read the contents of a file:

2.5.1 The read() method

This method is used to read a specified number of bytes of data from a data file. The syntax of read() method is:

```
file_object.read(n)
```

Consider the following set of statements to understand the usage of read() method:

```
>>>myobject=open("myfile.txt",'r')
>>> myobject.read(10)
'Hello ever'
>>> myobject.close()
```

If no argument or a negative number is specified in read(), the entire file content is read. For example,

```
>>> myobject=open("myfile.txt",'r')
>>> print(myobject.read())
Hello everyone
Writing multiline strings
This is the third line
>>> myobject.close()
```


Reprint 2025-26

2.5.2 The readline([n]) method

This method reads one complete line from a file where each line terminates with a newline (\n) character. It can also be used to read a specified number (n) of bytes of data from a file but maximum up to the newline character (\n). In the following example, the second statement reads the first ten characters of the first line of the text file and displays them on the screen.

```
>>> myobject=open("myfile.txt",'r')
>>> myobject.readline(10)
'Hello ever'
>>> myobject.close()
```

If no argument or a negative number is specified, it reads a complete line and returns string.

```
>>>myobject=open("myfile.txt",'r')
>>> print (myobject.readline())
'Hello everyone\n'
```

Activity 2.4

Create a file having multiline data and use readline() with an iterator to read the contents of the file line by line

To read the entire file line by line using the readline(), we can use a loop. This process is known as looping/ iterating over a file object. It returns an empty string when EOF is reached.

2.5.3 The readlines() method

The method reads all the lines and returns the lines along with newline as a list of strings. The following example uses readlines() to read data from the text file *myfile.txt*.

```
>>> myobject=open("myfile.txt", 'r')
>>> print(myobject.readlines())
['Hello everyone\n', 'Writing multiline
strings\n', 'This is the third line']
>>> myobject.close()
```

As shown in the above output, when we read a file using readlines() function, lines in the file become members of a list, where each list element ends with a newline character ('\n').

In case we want to display each word of a line separately as an element of a list, then we can use split() function. The following code demonstrates the use of split() function.

```
>>> myobject=open("myfile.txt",'r')
>>> d=myobject.readlines()
```


Reprint 2025-26

```
>>> for line in d:
        words=line.split()
        print(words)

['Hello', 'everyone']
['Writing', 'multiline', 'strings']
['This', 'is', 'the', 'third', 'line']
```

In the output, each string is returned as elements of a list. However, if *splitlines()* is used instead of split(), then each line is returned as element of a list, as shown in the output below:

```
>>> for line in d:
        words=line.splitlines()
        print(words)

['Hello everyone']
['Writing multiline strings']
['This is the third line']
```

Let us now write a program that accepts a string from the user and writes it to a text file. Thereafter, the same program reads the text file and displays it on the screen.

Program 2-1 Writing and reading to a text file

```
fobject=open("testfile.txt","w")    # creating a data file
sentence=input("Enter the contents to be written in the file: ")
fobject.write(sentence)             # Writing data to the file
fobject.close()                     # Closing a file

print("Now reading the contents of the file: ")
fobject=open("testfile.txt","r")
#looping over the file object to read the file
for str in fobject:
        print(str)
fobject.close()
```

In Program 2.1, the file named *testfile.txt* is opened in write mode and the file handle named *fobject* is returned. The string is accepted from the user and written in the file using write(). Then the file is closed and again opened in read mode. Data is read from the file and displayed till the end of file is reached.

Reprint 2025-26

Output of Program 2-1:

```
>>>
 RESTART: Path_to_file\Program2-1.py
Enter the contents to be written in the file:
roll_numbers = [1, 2, 3, 4, 5, 6]
Now reading the contents of the file:
roll_numbers = [1, 2, 3, 4, 5, 6]
>>>
```

2.6 Setting Offsets in a File

The functions that we have learnt till now are used to access the data sequentially from a file. But if we want to access data in a random fashion, then Python gives us seek() and tell() functions to do so.

2.6.1 The tell() method

This function returns an integer that specifies the current position of the file object in the file. The position so specified is the byte position from the beginning of the file till the current position of the file object. The syntax of using tell() is:

```
file_object.tell()
```

2.6.2 The seek() method

This method is used to position the file object at a particular position in a file. The syntax of seek() is:

```
file_object.seek(offset [, reference_point])
```

Think and Reflect

Does the seek() function work in the same manner for text and binary files?

In the above syntax, offset is the number of bytes by which the file object is to be moved. `reference_point` indicates the starting position of the file object. That is, with reference to which position, the offset has to be counted. It can have any of the following values:

0 - beginning of the file
1 - current position of the file
2 - end of file

By default, the value of `reference_point` is 0, i.e. the offset is counted from the beginning of the file. For example, the statement `fileObject.seek(5,0)` will position the file object at 5th byte position from the beginning of the file. The code in Program 2-2 below demonstrates the usage of seek() and tell().

Reprint 2025-26

Program 2-2 Application of seek() and tell()

```
print("Learning to move the file object")
fileobject=open("testfile.txt","r+")
str=fileobject.read()
print(str)
print("Initially, the position of the file object is: ",fileobject.
tell())
fileobject.seek(0)
print("Now the file object is at the beginning of the file:
",fileobject.tell())
fileobject.seek(10)
print("We are moving to 10th byte position from the beginning of
file")
print("The position of the file object is at", fileobject.tell())
str=fileobject.read()
print(str)
```

Output of Program 2-2:

```
>>>
 RESTART: Path_to_file\Program2-2.py
Learning to move the file object
roll_numbers = [1, 2, 3, 4, 5, 6]
Initially, the position of the file object is: 33
Now the file object is at the beginning of the file: 0
We are moving to 10th byte position from the beginning of file

The position of the file object is at 10
rs = [1, 2, 3, 4, 5, 6]
>>>
```

2.7 Creating and Traversing a Text File

Having learnt various methods that help us to open and close a file, read and write data in a text file, find the position of the file object and move the file object at a desired location, let us now perform some basic operations on a text file. To perform these operations, let us assume that we will be working with practice.txt.

2.7.1 Creating a file and writing data

To create a text file, we use the `open()` method and provide the filename and the mode. If the file already exists with the same name, the `open()` function will behave differently depending on the mode (write or append) used. If it is in write mode (w), then all the existing contents of file will be lost, and an empty file will be created with the same name. But, if the file is

Reprint 2025-26

created in append mode (a), then the new data will be written after the existing data. In both cases, if the file does not exist, then a new empty file will be created. In Program 2-3, a file, *practice.txt* is opened in write (w) mode and three sentences are stored in it as shown in the output screen that follows it

Program 2-3 To create a text file and write data in it

```
# program to create a text file and add data
fileobject=open("practice.txt","w+")
while True:
    data= input("Enter data to save in the text file: ")
    fileobject.write(data)
    ans=input("Do you wish to enter more data?(y/n): ")
    if ans=='n': break
fileobject.close()
```

Output of Program 2-3:

```
>>>
 RESTART: Path_to_file\Program2-3.py
Enter data to save in the text file: I am interested to learn about
Computer Science
Do you wish to enter more data?(y/n): y
Enter data to save in the text file: Python is easy to learn
Do you wish to enter more data?(y/n): n
>>>
```

2.7.2 Traversing a file and displaying data

To read and display data that is stored in a text file, we will refer to the previous example where we have created the file *practice.txt.* The file will be opened in read mode and reading will begin from the beginning of the file.

Program 2-4 To display data from a text file

```
fileobject=open("practice.txt","r")
str = fileobject.readline()
while str:
    print(str)
    str=fileobject.readline()
fileobject.close()
```

In Program 2-4, the readline() is used in the while loop to read the data line by line from the text file. The lines are displayed using the print(). As the end of file is reached, the readline() will return an empty string. Finally, the file is closed using the close().

Output of Program 2-4:

```
>>>
I am interested to learn about Computer SciencePython is easy to learn
```

Till now, we have been creating separate programs for writing data to a file and for reading the file. Now let us create one single program to read and write data using a single file object. Since both the operations have to be performed using a single file object, the file will be opened in w+ mode.

Program 2-5 To perform reading and writing operation in a text file

```
fileobject=open("report.txt", "w+")
print ("WRITING DATA IN THE FILE")
print() # to display a blank line
while True:
    line= input("Enter a sentence ")
    fileobject.write(line)
    fileobject.write('\n')
    choice=input("Do you wish to enter more data? (y/n): ")
    if choice in ('n','N'): break
print("The byte position of file object is ",fileobject.tell())
fileobject.seek(0)  #places file object at beginning of file
print()
print("READING DATA FROM THE FILE")
str=fileobject.read()
print(str)
fileobject.close()
```

In Program 2-5, the file will be read till the time end of file is not reached and the output as shown in below is displayed.

Output of Program 2-5:

```
>>>
 RESTART: Path_to_file\Program2-5.py
WRITING DATA IN THE FILE

Enter a sentence I am a student of class XII
Do you wish to enter more data? (y/n): y
Enter a sentence my school contact number is 4390xxx8
Do you wish to enter more data? (y/n): n
The byte position of file object is  67

READING DATA FROM THE FILE
I am a student of class XII
my school contact number is 4390xxx8
>>>
```


Reprint 2025-26

2.8 THE PICKLE MODULE

We know that Python considers everything as an object. So, all data types including list, tuple, dictionary, etc. are also considered as objects. During execution of a program, we may require to store current state of variables so that we can retrieve them later to its present state. Suppose you are playing a video game, and after some time, you want to close it. So, the program should be able to store the current state of the game, including current level/stage, your score, etc. as a Python object. Likewise, you may like to store a Python dictionary as an object, to be able to retrieve later. To save any object structure along with data, Python provides a module called Pickle. The module Pickle is used for serializing and de-serializing any Python object structure. Pickling is a method of preserving food items by placing them in some solution, which increases the shelf life. In other words, it is a method to store food items for later consumption.

Serialization is the process of transforming data or an object in memory (RAM) to a stream of bytes called byte streams. These byte streams in a binary file can then be stored in a disk or in a database or sent through a network. Serialization process is also called pickling.

De-serialization or unpickling is the inverse of pickling process where a byte stream is converted back to Python object.

The pickle module deals with binary files. Here, data are not written but dumped and similarly, data are not read but loaded. The Pickle Module must be imported to load and dump data. The pickle module provides two methods - dump() and load() to work with binary files for pickling and unpickling, respectively.

2.8.1 The dump() method

This method is used to convert (pickling) Python objects for writing data in a binary file. The file in which data are to be dumped, needs to be opened in binary write mode (wb).

Syntax of `dump()` is as follows:

```
dump(data_object, file_object)
```

where `data_object` is the object that has to be dumped to the file with the file handle named `file_`

object. For example, Program 2-6 writes the record of a student (roll_no, name, gender and marks) in the binary file named mybinary.dat using the dump(). We need to close the file after pickling.

Program 2-6 Pickling data in Python

```
import pickle
listvalues=[1,"Geetika",'F', 26]
fileobject=open("mybinary.dat", "wb")
pickle.dump(listvalues,fileobject)
fileobject.close()
```

2.8.2 The load() method

This method is used to load (unpickling) data from a binary file. The file to be loaded is opened in binary read (rb) mode. Syntax of load() is as follows:

Store_object = load(file_object)

Here, the pickled Python object is loaded from the file having a file handle named file_object and is stored in a new file handle called store_object. The program 2-7 demonstrates how to read data from the file *mybinary.dat* using the load().

Program 2-7 Unpickling data in Python

```
import pickle
print("The data that were stored in file are: ")
fileobject=open("mybinary.dat","rb")
objectvar=pickle.load(fileobject)
fileobject.close()
print(objectvar)
```

Output of Program 2-7:

```
>>>
 RESTART: Path_to_file\Program2-7.py
The data that were stored in file are:
[1, 'Geetika', 'F', 26]
>>>
```

2.8.3 File handling using pickle module

As we read and write data in a text file, similarly we will be adding and displaying data for a binary file. Program 2-8 accepts a record of an employee from the user and appends it in the binary file *tv*. Thereafter, the records are read from the binary file and displayed on the screen using the same object. The user may enter

Reprint 2025-26

as many records as they wish to. The program also displays the size of binary files before starting with the reading process.

Program 2-8 To perform basic operations on a binary file using pickle module

```
# Program to write and read employee records in a binary file
import pickle
print("WORKING WITH BINARY FILES")
bfile=open("empfile.dat","ab")
recno=1
print ("Enter Records of Employees")
print()
#taking data from user and dumping in the file as list object
while True:
    print("RECORD No.", recno)
    eno=int(input("\tEmployee number : "))
    ename=input("\tEmployee Name : ")
    ebasic=int(input("\tBasic Salary : "))
    allow=int(input("\tAllowances : "))
    totsal=ebasic+allow
    print("\tTOTAL SALARY : ", totsal)
    edata=[eno,ename,ebasic,allow,totsal]
    pickle.dump(edata,bfile)
    ans=input("Do you wish to enter more records (y/n)? ")
    recno=recno+1
    if ans.lower()=='n':
        print("Record entry OVER ")
        print()
        break
# retrieving the size of file
print("Size of binary file (in bytes):",bfile.tell())
bfile.close()
# Reading the employee records from the file using load() module
print("Now reading the employee records from the file")
print()
readrec=1
try:
  with open("empfile.dat","rb") as bfile:
  while True:
       edata=pickle.load(bfile)
       print("Record Number : ",readrec)
       print(edata)
       readrec=readrec+1
except EOFError:
    pass
bfile.close()
```

Output of Program 2-8:

```
>>>
 RESTART: Path_to_file\Program2-8.py
WORKING WITH BINARY FILES
Enter Records of Employees

RECORD No. 1
        Employee number : 11
        Employee Name : D N Ravi
        Basic Salary : 32600
        Allowances : 4400
        TOTAL SALARY :  37000
Do you wish to enter more records (y/n)? y
RECORD No. 2
        Employee number : 12
        Employee Name : Farida Ahmed
        Basic Salary : 38250
        Allowances : 5300
        TOTAL SALARY :  43550
Do you wish to enter more records (y/n)? n
Record entry OVER

Size of binary file (in bytes): 216
Now reading the employee records from the file

Record Number :  1
[11, 'D N Ravi', 32600, 4400, 37000]
Record Number :  2
[12, 'Farida Ahmed', 38250, 5300, 43550]
>>>
```

As each employee record is stored as a list in the file `empfile.dat`, hence while reading the file, a list is displayed showing record of each employee. Notice that in Program 2-8, we have also used try.. except block to handle the end-of-file exception.

SUMMARY

- A file is a named location on a secondary storage media where data are permanently stored for later access.
- A text file contains only textual information consisting of alphabets, numbers and other

Reprint 2025-26

special symbols. Such files are stored with extensions like *.txt, .py, .c, .csv, .html,* etc. Each byte of a text file represents a character.

- Each line of a text file is stored as a sequence of ASCII equivalent of the characters and is terminated by a special character, called the End of Line (EOL).
- Binary file consists of data stored as a stream of bytes.
- open() method is used to open a file in Python and it returns a file object called file handle. The file handle is used to transfer data to and from the file by calling the functions defined in the Python's io module.
- close() method is used to close the file. While closing a file, the system frees up all the resources like processor and memory allocated to it.
- `write()` method takes a string as an argument and writes it to the text file.
- `writelines()` method is used to write multiple strings to a file. We need to pass an iterable object like lists, tuple etc. containing strings to writelines() method.
- `read([n])` method is used to read a specified number of bytes (n) of data from a data file.
- `readline([n])` method reads one complete line from a file where lines are ending with a newline (\n). It can also be used to read a specified number (n) of bytes of data from a file but maximum up to the newline character (\n).
- `readlines()` method reads all the lines and returns the lines along with newline character, as a list of strings.
- `tell()` method returns an integer that specifies the current position of the file object. The position so specified is the byte position from the beginning of the file till the current position of the file object.
- `seek()` method is used to position the file object at a particular position in a file.

Reprint 2025-26

- Pickling is the process by which a Python object is converted to a byte stream.
- `dump()` method is used to write the objects in a binary file.
- `load()` method is used to read data from a binary file.

EXERCISE

1. Differentiate between:
   a) text file and binary file
   b) readline() and readlines()
   c) write() and writelines()

2. Write the use and syntax for the following methods:
   a) open()
   b) read()
   c) seek()
   d) dump()

3. Write the file mode that will be used for opening the following files. Also, write the Python statements to open the following files:
   a) a text file "example.txt" in both read and write mode
   b) a binary file "bfile.dat" in write mode
   c) a text file "try.txt" in append and read mode
   d) a binary file "btry.dat" in read only mode.

4. Why is it advised to close a file after we are done with the read and write operations? What will happen if we do not close it? Will some error message be flashed?

5. What is the difference between the following set of statements (a) and (b):
   a) P = open("practice.txt","r")
      P.read(10)
   b) with open("practice.txt", "r") as P:
         x = P.read()

6. Write a command(s) to write the following lines to the text file named *hello.txt*. Assume that the file is opened in append mode.
   " Welcome my class"
   "It is a fun place"
   "You will learn and play"

Reprint 2025-26

7. Write a Python program to open the file *hello.txt* used in question no 6 in read mode to display its contents. What will be the difference if the file was opened in write mode instead of append mode?

8. Write a program to accept string/sentences from the user till the user enters "END" to. Save the data in a text file and then display only those sentences which begin with an uppercase alphabet.

9. Define pickling in Python. Explain serialization and deserialization of Python object.

10. Write a program to enter the following records in a binary file:

Item No	integer
Item_Name	string
Qty	integer
Price	float

Number of records to be entered should be accepted from the user. Read the file to display the records in the following format:

Item No:
Item Name :
Quantity:
Price per item:
Amount: (to be calculated as Price * Qty)

Reprint 2025-26

Chapter 3 Stack

12130CH03

"We're going to be able to ask our computers to monitor things for us, and when certain conditions happen, are triggered, the computers will take certain actions and inform us after the fact."

— *Steve Jobs*

In this Chapter

3.1 Introduction

We have learnt about different data types in Python for handling values in Class XI. Recall that String, List, Set, Tuple, etc. are the sequence data types that can be used to represent collection of elements either of the same type or different types. Multiple data elements are grouped in a particular way for faster accessibility and efficient storage of data. That is why we have used different data types in python for storing data values. Such grouping is referred as a data structure.

A data structure defines a mechanism to store, organise and access data along with operations (processing) that can be efficiently performed on the data. For example, string is a data structure containing a sequence of elements where each element is a character. On the other hand, list is a sequence data structure in which each element may be of different types. We can apply different operations like reversal, slicing, counting of

elements, etc. on list and string. Hence, a data structure organises multiple elements in a way so that certain operations on each element as well as the collective data unit could be performed easily.

Stack and *Queue* are two other popular data structures used in programming. Although not directly available in Python, it is important to learn these concepts as they are extensively used in a number of programming languages. In this chapter, we will study about stack, its implementation using Python as well as its applications.

Other important data structures in Computer Science include Array, Linked List, Binary Trees, Heaps, Graph, Sparse Matrix, etc.

3.2 Stack

We have seen piles of books in the library or stack of plates at home (Figure 3.1). To put another book or another plate in such a pile, we always place (add to the pile) the object at the top only. Likewise, to remove a book or a plate from such a pile, we always remove (delete from the pile) the object from the top only. This is because in a large pile, it is inconvenient to add or remove an object from in between or bottom. Such an arrangement of elements in a linear order is called a stack. We add new elements or remove existing elements from the same end, commonly referred to as the *top* of the stack. It thus follows the Last-In-First-out (LIFO) principle. That is, the element which was inserted last (the most recent element) will be the first one to be taken out from the stack.

A data structure in which elements are organised in a sequence is called linear data structure.

Figure 3.1: Stack of plates and books

3.2.1 APPLICATIONS OF STACK

Some of the applications of stack in real-life are:

- Pile of clothes in an almirah
- Multiple chairs in a vertical pile
- Bangles worn on wrist
- Pile of boxes of eatables in pantry or on a kitchen shelf

Some examples of application of stack in programming are as follows:

- When we need to reverse a string, the string is traversed from the last character till the first character. i.e. characters are traversed in the reverse order of their appearance in the string. This is very easily done by putting the characters of a string in a stack.
- We use text/image editor for editing the text/image where we have options to redo/undo the editing done. When we click on the redo /undo icon, the most recent editing is redone/undone. In this scenario, the system uses a stack to keep track of changes made.
- While browsing the web, we move from one web page to another by accessing links between them. In order to go back to the last visited web page, we may use the back button on the browser. Let us say we accessed a web page P1 from where we moved to web page P2 followed by browsing of web page P3. Currently, we are on web page P3 and want to revisit web page P1. We may go to a previously visited web page by using the BACK button of the browser. On clicking the BACK button once, we are taken from web page P3 to web page P2, another click on BACK shows web page P1. In this case, the history of browsed pages is maintained as stack.
- While writing any arithmetic expression in a program, we may use parentheses to order the evaluation of operators. While executing the program, the compiler checks for matched parentheses i.e. each opening parenthesis should have a corresponding closing parenthesis and the pairs of parentheses are properly nested. In case of parentheses are

Think and Reflect

How does a compiler or an interpreter handle function calls in a program?

Think and Reflect

The operating system in computer or mobile allocates memory to different applications for their execution. How does an operating system keep track of the free memory that can be allocated among programs/ applications to be executed?

mismatched, the compiler needs to throw an error. To handle matching of parentheses, stack is used.

3.3 Operations on Stack

As explained in the previous section, a stack is a mechanism that implements LIFO arrangement hence elements are added and deleted from the stack at one end only. The end from which elements are added or deleted is called TOP of the stack. Two fundamental operations performed on the stack are PUSH and POP. In this section, we will learn about them and implement them using Python.

3.3.1 PUSH and POP Operations

- PUSH adds a new element at the TOP of the stack. It is an insertion operation. We can add elements to a stack until it is full. A stack is full when no more elements can be added to it. Trying to add an element to a full stack results in an exception called 'overflow'.
- POP operation is used to remove the top most element of the stack, that is, the element at the TOP of the stack. It is a delete operation. We can delete elements from a stack until it is empty i.e. there is no element in it. Trying to delete an element from an empty stack results in an exception called 'underflow'.

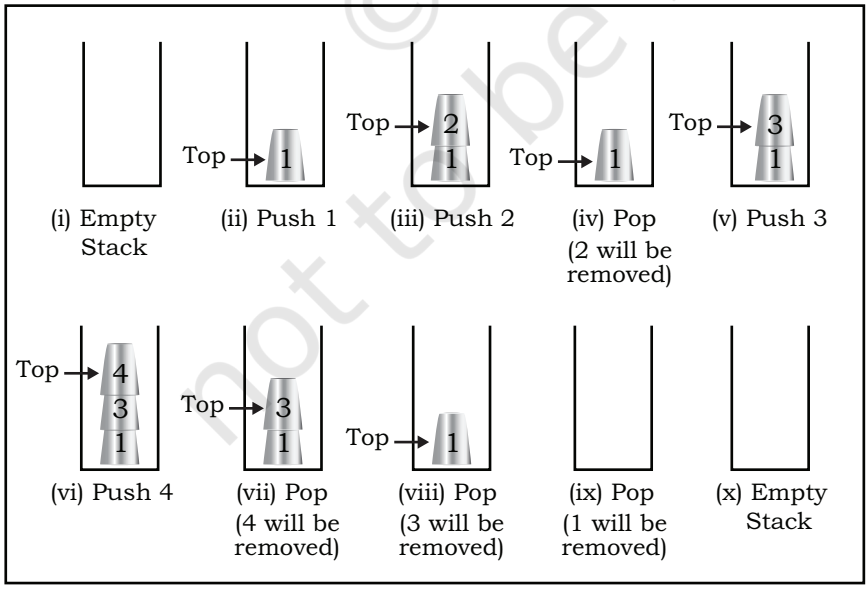


Figure 3.2: PUSH and POP operations on the stack of glasses

A stack is used to insert and delete elements in LIFO order. Same principle is followed in adding and removing glasses from a pile of glasses. Let us create a stack of glasses assuming that each glass is numbered. Visual representations of PUSH and POP operations on a stack of glasses are shown in Figure 3.2.

Reprint 2025-26

3.4 IMPLEMENTATION OF STACK IN PYTHON

We have learnt so far that a stack is a linear and ordered collection of elements. The simple way to implement a stack in Python is using the data type *list*. We can fix either of the sides of the list as TOP to insert/remove elements. It is to be noted that we are using built-in methods append() and pop() of the list for implementation of the stack. As these built-in methods insert/delete elements at the rightmost end of the list, hence explicit declaration of TOP is not needed.

Let us write a program to create a STACK (stack of glasses as given in Figure 3.2) in which we will:

- insert/delete elements (glasses)
- check if the STACK is empty (no glasses in the stack)
- find the number of elements (glasses) in the STACK
- read the value of the topmost element (number on the topmost glass) in the STACK

The program shall define the following functions to perform these operations:

- Let us create an empty stack named `glassStack`. We will do so by assigning an empty list to the identifier named `glassStack`:

```
glassStack = list()
```

- A function named isEmpty to check whether the stack `glassStack` is empty or not. Remember trying to remove an element from an empty stack would result in 'underflow'. This function returns True if the stack is empty, else returns False.

```
def isEmpty(glassStack):
    if len(glassStack)==0:
        return True
    else:
        return False
```

- A function named opPush to insert (`PUSH`) a new element in stack. This function has two parameters - the name of the stack in which the element is to be inserted (`glassStack`) and the element that needs to be inserted. We know that insertion of an element is always done at the TOP of the stack. Hence, we

Reprint 2025-26

shall use the built-in method `append()` of list to add an element to the stack that always adds at the end of the list. As there is no limit on the size of list in Python, the implemented stack will never be full unless there is no more space available in memory. Hence, we will never face 'overflow' (no space for new element) condition for stack.

```
def opPush(glassStack,element):
    glassStack.append(element)
```

- A function named `size` to read the number of elements in the `glassStack`. We will use the len() function of list in Python to find the number of elements in the `glassStack`.

```
def size(glassStack):
    return len(glassStack)
```

- A function named top to read the most recent element (`TOP`) in the `glassStack`.

```
def top(glassStack):
    if isEmpty(glassStack):
        print('Stack is empty')
        return None
    else:
        x =len(glassStack)
        element=glassStack[x-1]
        return element
```

- A function named `opPop` to delete the topmost element from the stack. It takes one parameter - the name of the stack (`glassStack`) from which element is to be deleted and returns the value of the deleted element. The function first checks whether the stack is empty or not. If it is not empty, it removes the topmost element from it. We shall use the built-in method `pop()` of Python list that removes the element from the end of the list.

```
def opPop(glassStack):
    if isEmpty(glassStack):
        print('underflow')
        return None
    else:
        return(glassStack.pop())
```

- A function named display to show the contents of the stack.

```
def display(glassStack):
    x=len(glassStack)
    print("Current elements in the stack
are: ")
    for i in range(x-1,-1,-1):
        print(glassStack[i])
```

Once we define the above functions we can use the following Python code to implement a stack of glasses.

```
glassStack = list()  # create empty stack

#add elements to stack
element='glass1'
print("Pushing element ",element)
opPush(glassStack,element)
element='glass2'
print("Pushing element ",element)
opPush(glassStack,element)

#display number of elements in stack
print("Current number of elements in stack
is",size(glassStack))

#delete an element from the stack
element=opPop(glassStack)
print("Popped element is",element)

#add new element to stack
element='glass3'
print("Pushing element ",element)
opPush(glassStack,element)

#display the last element added to the
#stack
print("top element is",top(glassStack))

#display all elements in the stack
display(glassStack)
```


Reprint 2025-26

```
#delete all elements from stack
while True:
   item=opPop(glassStack)
   if item == None:
      print("Stack is empty now")
      break
   else:
      print("Popped element is",item)
```

The output of the above program will be as follows:

```
Pushing element  glass1
Pushing element  glass2
Current number of elements in stack is 2
Popped element is glass2
Pushing element  glass3
top element is glass3
Current elements in the stack are:
glass3
glass1
Popped element is glass3
Popped element is glass1
Underflow
Stack is empty now
```

3.5 NOTATIONS FOR ARITHMETIC EXPRESSIONS

We write arithmetic expressions using operators in between operands, like x + y, 2 - 3 * y, etc. and use parentheses () to order the evaluation of operators in complex expressions. These expressions follow infix representation and are evaluated using BODMAS rule.

Polish mathematician Jan Lukasiewicz in the 1920's introduced a different way of representing arithmetic expression, called polish notation. In such notation, operators are written before their operands. So the order of operations and operands determines the result, making parentheses unnecessary. For example, we can write x+y in polish notation as +xy. This is also called prefix notation as we prefix the operator before operands.

By reversing this logic, we can write an expression by putting operators after their operands. For example, x+y can be written as xy+. This is called reverse polish

Reprint 2025-26

notation or postfix notation. To summarise, any arithmetic expression can be represented in any of the three notations viz. Infix, Prefix and Postfix and are listed in Table 3.1 with examples.

Table 3.1 Infix, Prefix and Postfix Notations

Type of Expression	Description	Example
Infix	Operators are placed in between the operands	x * y + z 3 *(4 + 5) (x + y)/(z * 5)
Prefix (Polish)	Operators are placed before the corresponding operands	+z*xy *3+45 /+xy*z5
Postfix (Reverse Polish)	Operators are placed after the corresponding operands	xy*z+ 345+* xy+z5*/

3.6 Conversion from Infix to Postfix Notation

It is easy for humans to evaluate an *infix* expression. Consider an *infix* expression x + y / z. While going from left to right we first encounter + operator, but we do not add x + y and rather evaluate y/z, followed by addition operation. This is because we know the order of precedence of operators that follows BODMAS rule. But, how do we pass this precedence knowledge to the computer through an expression?

In contrast, *prefix/postfix* expressions do not have to deal with such precedence because the operators are already positioned according to their order of evaluation. Hence a single traversal from left to right is sufficient to evaluate the expression. In this section, we will learn about the conversion of an arithmetic expression written in *infix* notation to its equivalent expression in *postfix* notation using a stack.

During such conversion, a stack is used to keep track of the operators encountered in the *infix* expression. A variable of string type is used to store the equivalent *postfix* expression. Algorithm 3.1 converts an expression in *infix* notation to *postfix* notation:

Think and Reflect

Write an algorithm to convert an infix expression into equivalent prefix expression using stack.

Algorithm 3.1: Conversion of expression from infix to postfix notation

Step 1: Create an empty string named postExp to store the converted postfix expression.
Step 2: INPUT infix expression in a variable, say inExp
Step 3: For each character in inExp, REPEAT Step 4

Reprint 2025-26

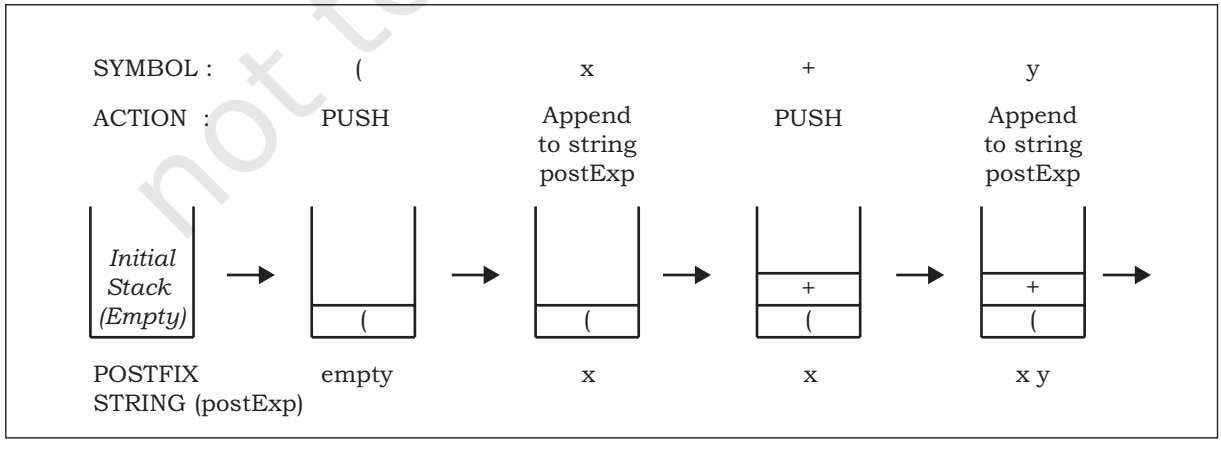
Step 4: IF character is a left parenthesis THEN PUSH on the Stack
ELSE IF character is a right parenthesis
THEN POP the elements from the Stack and append to string postExp until the corresponding left parenthesis is popped while discarding both left and right parentheses
ELSE IF character is an operator
THEN IF its precedence is lower than that of operator at the top of Stack
THEN POP elements from the Stack till an operator with precedence less than the current operator is encountered and append to string postExp before pushing this operator on the postStack
ELSE PUSH operator on the Stack
ELSE Append the character to postExp

Step 5: Pop elements from the Stack and append to postExp until Stack is empty

Step 6: OUTPUT postExp

Example 3.1

Let us now use this algorithm to convert a given infix expression (x + y) / (z*8) into equivalent postfix expression using a stack. Figure 3.3 shows the steps to be followed on encountering an operator or an operand in the given infix expression. Note here that stack is used to track the operators and parentheses, and a string variable contains the equivalent postfix expression. Initially both are empty. Each character in the given infix expression is processed from left to right and the appropriate action is taken as detailed in the algorithm. When each character in the given infix expression has been processed, the string will contain the equivalent postfix expression.



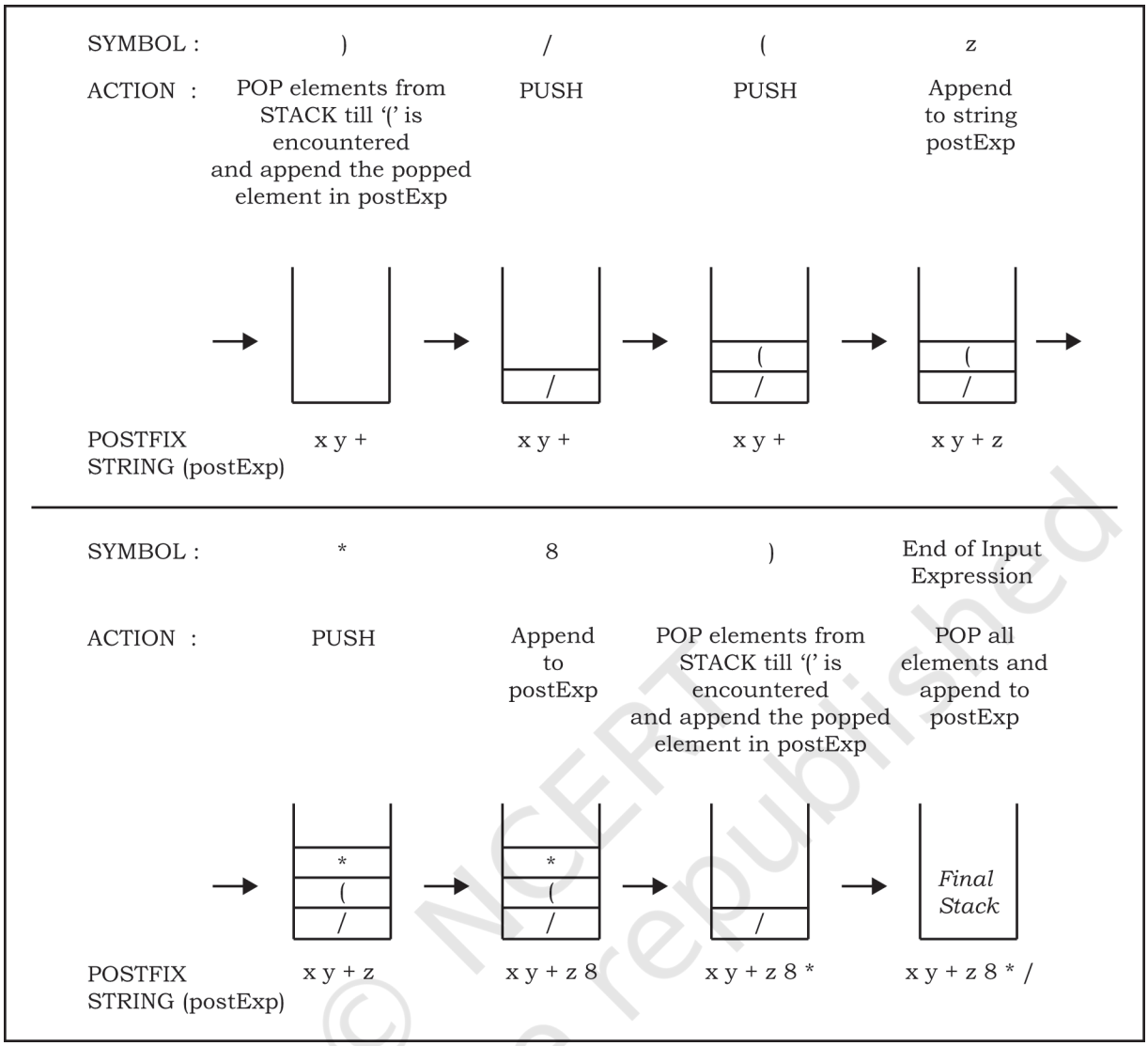


*Figure 3.3: Conversion of infix expression (x + y)/(z*8) to postfix notation*

3.7 Evaluation of Postfix Expression

Stacks can be used to evaluate an expression in postfix notation. For simplification, we are assuming that operators used in expressions are binary operators. The detailed step by step procedure is given in Algorithm 3.2.

Algorithm 3.2: Evaluation of postfix expression

Step 1: INPUT postfix expression in a variable, say postExp

Step 2: For each character in postExp, REPEAT Step 3

Step 3: IF character is an operand

THEN PUSH character on the Stack

ELSE POP two elements from the Stack, apply the operator on

the popped elements and PUSH the computed value onto the Stack

Step 4: IF Stack has a single element

THEN POP the element and OUTPUT as the net result

ELSE OUTPUT "Invaild Postfix expression"

Example 3.2

Figure 3.4 shows the step-by-step process of evaluation of the postfix expression 7 8 2 * 4 / + using Algorithm 3.2 .

Think and Reflect

Write an algorithm to evaluate any prefix expression using a stack.

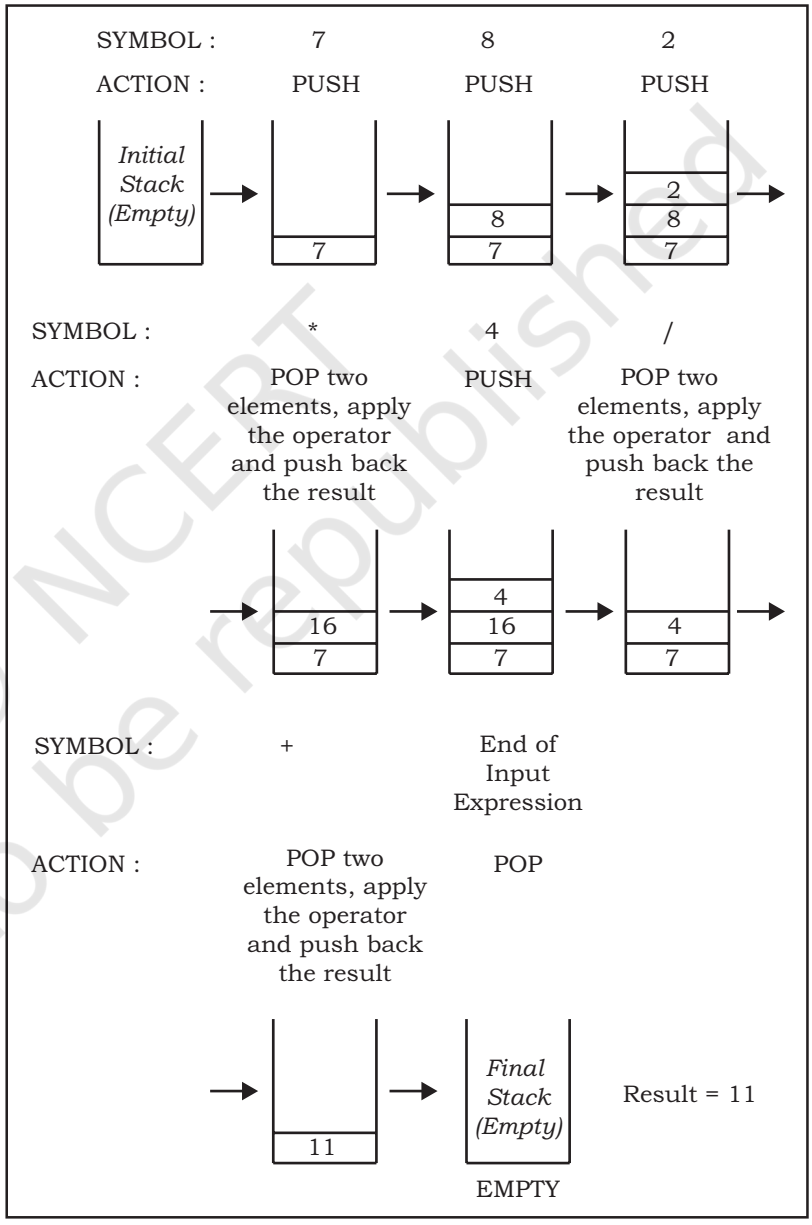


*Figure 3.4: Evaluation of postfix expression 7 8 2 * 4 / +*

Reprint 2025-26

SUMMARY

- Stack is a data structure in which insertion and deletion is done from one end only, usually referred to as TOP.
- Stack follows LIFO principle using which an element inserted in the last will be the first one to be out.
- PUSH and POP are two basic operations performed on a stack for insertion and deletion of elements, respectively.
- Trying to pop an element from an empty stack results into a special condition *underflow.*
- In Python, list is used for implementing a stack and its built-in-functions *append* and *pop* are used for insertion and deletion, respectively. Hence, no explicit declaration of TOP is needed.
- Any arithmetic expression can be represented in any of the three notations viz. Infix, Prefix and Postfix.
- While programming, Infix notation is used for writing an expression in which binary operators are written in between the operands.
- A single traversal from left to right of Prefix/ Postfix expression is sufficient to evaluate the expression as operators are correctly placed as per their order of precedence.
- Stack is commonly used data structure to convert an Infix expression into equivalent Prefix/Postfix notation.
- While conversion of an Infix notation to its equivalent Prefix/Postfix notation, only operators are PUSHed onto the Stack.
- When evaluating any Postfix expression using Stack, only operands are PUSHed onto it.

Reprint 2025-26

EXERCISE

1. State TRUE or FALSE for the following cases:
   a) Stack is a linear data structure
   b) Stack does not follow LIFO rule
   c) PUSH operation may result into underflow condition
   d) In POSTFIX notation for expression, operators are placed after operands

2. Find the output of the following code:
   a)
```
result=0
numberList=[10,20,30]
numberList.append(40)
result=result+numberList.pop()
result=result+numberList.pop()
print("Result=",result)
```
   b)
```
answer=[]; output=''
answer.append('T')
answer.append('A')
answer.append('M')
ch=answer.pop()
output=output+ch
ch=answer.pop()
output=output+ch
ch=answer.pop()
output=output+ch
print("Result=",output)
```

3. Write a program to reverse a string using stack.

4. For the following arithmetic expression:
   ((2+3)*(4/2))+2
   Show step-by-step process for matching parentheses using stack data structure.

5. Evaluate following postfix expressions while showing status of stack after each operation given A=3, B=5, C=1, D=4
   a) A B + C *
   b) A B * C / D *

6. Convert the following infix notations to postfix notations, showing stack and string contents at each step.
   a) A + B - C * D
   b) A * ((C + D)/E)

7. Write a program to create a Stack for storing only odd numbers out of all the numbers entered by the user. Display the content of the Stack along with the largest odd number in the Stack. (Hint. Keep popping out the elements from stack and maintain the largest element retrieved so far in a variable. Repeat till Stack is empty)