

CISCO NETWORKING ACADEMY

PYTHON ESSENTIALS-1

SUBMITTED

by

K V K SAI BHASKAR

A21126510023

In the fulfilment of the Internship in

COMPUTER SCIENCE AND ENGINEERING



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(AFFILIATED BY ANDHRA UNIVERSITY)

SANGHIVALASA, VISAKHAPATNAM – 531162

2021-2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(AFFILIATED BY ANDHRA UNIVERSITY)

SANGHIVALASA, VISAKHAPATNAM – 531162

2021 – 2025

BONAFIDE CERTIFICATE

This is to certify that this Internship Report “Python Essentials” is the bonafide work of K.V.K.SAI BHASKAR (A21126510023) of III/IV CSE carried out Internship under my supervision.

Reviewer
Mr. B. Mahesh
Assistant Professor
Department of CSE
ANITS

Class Teacher
Mrs. Spandana Valli
Assistant Professor
Department of CSE

Head of The Department
Department of CSE
ANITS

ACKNOWLEDGEMENT

An endeavour over a long period can be successful with the advice and support of many well-wishers. We take this opportunity to express our gratitude and appreciation to all of them. We owe our tributes to Dr. P E S N Krishna Prasad , Head of the Department, Computer Science & Engineering, ANITS, for his valuable support and guidance during the period of the Internship.

We wish to express our sincere thanks and gratitude to our Course Curators Cisco Networking Academy who helped in stimulating discussions, and for guiding us throughout the Course. We express our warm and sincere thanks for the encouragement, untiring guidance, and confidence they had shown in us.

We also thank all the staff members of the Computer Science & Engineering department for their valuable advices. We also thank supporting staff for providing resources as and when required.

K V K SAI BHASKAR

A21126510023

TABLE OF CONTENTS

- Introduction to Python and Computer Programming
 - Welcome to Python Essentials 1
 - Section 1- Introduction to Programming
 - Section 2- Introduction to Python
 - Section 3- Downloading and Installing Python
 - Module 1 Completion- MODULE TEST
- Python Data types, Variables, Operators and Basic I/O Operations
 - Section 1- The “Hello, World!” Program
 - Section 2- Python Literals
 - Section 3- Operators-data manipulation tools
 - Section 4- Variables
 - Section 5- Comments
 - Section 6- Interaction with the user
 - Module 2 Completion- MODULE TEST
- Boolean Values, Conditional Execution, Loops, List and List Processing, Logical and Bitwise Operations
 - Section 1- Making decisions in Python
 - Section 2- Loops in Python
 - Section 3- Logic and bit operations in Python
 - Section 4- Lists
 - Section 5- Sorting simple lists: the bubble sort algorithm
 - Section 6- Operations on List
 - Section 7- Lists in advanced applications
 - Module 3 Completion- MODULE TEST
- Functions, Tuples, Dictionaries, Exceptions, and Data Processing
 - Section 1- Functions
 - Section 2- How functions communicate with their environment
 - Section 3- Returning a result from a function
 - Section 4- Scopes in Python
 - Section 5- Creating multi-parameter functions
 - Section 6- Tuples and dictionaries
 - Section 7- Exceptions
 - Module 4 Completion- MODULE TEST
- Conclusion

MODULE 1

INTRODUCTION TO PYTHON AND COMPUTER PROGRAMMING

Greetings from the fascinating world of programming with the ever-flexible Python language! The capacity to comprehend and create code is becoming more and more important in today's technologically driven world. Acquiring knowledge in Python may lead to countless opportunities in the field of computer programming, regardless of your level of experience.

Fundamentally, programming is the art of giving instructions to a computer to carry out tasks. It's similar to teaching someone a new language, but you're speaking with robots rather than people. Python is an excellent option for novices as it is a high-level and adaptable programming language. It's a great place to start for anyone interested in learning about coding because of its readability and simplicity.

→ Welcome to Python Essentials 1

Programming languages are an essential component of the ever-changing technological environment because they influence how we use computers and create applications. Python is one language that has endured and is still quite popular in the programming community. Python is the preferred language for both novice and experienced developers due to its popularity, readability, and ease of use. Python is not only a language of today; as we explore its complexities, it becomes clear that it is also a language of the future, bringing with it boundless potential and inventiveness.

Python stands out due to its intuitive syntax, which enables programmers to convey complicated concepts in a few of lines of code. Python supports a wide range of applications, whether you are a beginner playing with your first "Hello, World!" program or an expert coder working on complex machine learning techniques. Scientific computing libraries like NumPy and SciPy, web development frameworks like Django and Flask, and data analysis tools like Pandas are examples of its adaptability. Python is a top option for developers looking for productivity and efficiency because of its smooth integration with other languages and technologies.

For those who are interested in learning Python programming, Cisco's "Python Essentials 1" course is a great starting point because it gives a strong foundation in the language.

Some of the main reasons why this course is very good:

- **Structured Learning Path:** The course is set up with a step-by-step progression from the fundamentals to increasingly complex ideas. This method makes sure that students have a solid foundation before tackling difficult subjects.
- **Extensive Curriculum:** Variables, data types, loops, functions, and object-oriented programming are just a few of the fundamental Python concepts that are covered in this course. It caters to both novices and those with some programming expertise by offering a thorough understanding of the language.
- **Interactive Learning:** The course frequently incorporates interactive components including tests, coding challenges, and practical laboratories. By enabling students to use their knowledge in the present, interactive learning improves comprehension by successfully reinforcing concepts.
- **Relevance to the Real World:** As a respectable technological corporation, Cisco makes sure that the course material is in line with practical uses. It focuses on real-world scenarios and examples so that students may see how Python is applied in the business world.
- **Excellent Instruction:** Video lectures, texts, and demonstrations are just a few of the excellent teaching resources that are frequently used in Cisco courses. The knowledge and skill of seasoned teachers is advantageous to students, improving their entire educational experience.

→SECTION 1—INTRODUCTION TO PROGRAMMING

In this section, we will mainly discuss about how does a program works, normal languages v/s Programming languages, Machine language v/s high-level language, Compilation v/s Interpretation, and many more.

How does a program work

Computer programs are the silent architects that weave the complex web of contemporary technology, creating the virtual environment we live in. Every software, smartphone application, and website are powered by a computer program—a painstakingly created collection of instructions. However, how can this clever invention turn random lines of code into engaging, dynamic experiences? You have to go through the worlds of logic, algorithms, and binary magic in order to comprehend the essence of how a computer program works.

A computer program is really just a set of instructions written in a programming language, which is a specially constructed vernacular that is understandable by both computers and humans. This code acts as a blueprint, describing the steps a computer must take in order to

do a certain task. These languages, which range from the sophisticated Python to the reliable Java, serve as middlemen, converting human reasoning into a form that the computer's processor can understand.

An algorithm is a fundamental component of every computer program. It is a methodical process or formula created to solve a given issue or carry out a certain activity. The program's algorithms are its brains; they precisely plan and direct the flow of activities. Algorithms offer the theoretical foundation upon which programs are constructed, whether they are used to sort a list of numbers, look for a specific item in a database, or simulate complicated systems.

Data is the lifeblood that flows through programs in the field of computer programming. Programs work with data, be it user-provided input, data extracted from databases, or sensor readings from the real environment. Through the processing, transformation, and storage of data, programs are able to produce outputs, make decisions, and adjust to changing conditions. Programs give life to unstructured data by utilizing variables, arrays, and intricate data structures to transform it into meaningful and practical information.

Normal Languages v/s Programming Languages

Human communication has always relied on languages in one manner or another. Languages are the means by which thoughts, feelings, and information are communicated. They can be spoken, written, or signed. There are several sorts of languages in the wide world of communication, each with a specialized function. The two different yet interrelated threads in the fabric of human expression that illustrate the duality between human communication and the language of machines are "normal" languages and programming languages.

Humans speak normal languages, also referred to as natural languages, to communicate with one another. These are the many languages spoken around the world, including English, Spanish, Mandarin, and many more. Natural languages are rich in context, cultural importance, and subtleties. With time, they change to meet the demands and adjust to the dynamics of human civilizations. Humans communicate, express emotions, reach agreements, and delve into the depths of literature and the arts through their native languages. These languages constitute the foundation of human understanding and connection because of their intrinsic flexibility, which enables the communication of a vast range of complex ideas and nuanced emotions.

Programming languages, on the other hand, are developed for a very different purpose: they are the mechanism by which people give instructions to computers. Programmers may build algorithms and develop software applications thanks to these organized, accurate, and clear languages. Programming languages like Java, C++, Python, and JavaScript are designed for human-to-machine communication rather than human-to-human communication. Programmers may create everything from straightforward scripts to

intricate, sophisticated software systems because to these principles and grammar, which specify how instructions should be expressed and understood.

Machine Level v/s High Level Language

Machine level language and high-level language are two opposing languages that form the basis of software creation in the broad field of computer programming. Within the computer domain, each language serves a variety of audiences and requirements at varying degrees of abstraction.

Programming's lowest level of abstraction is represented by machine level language, or machine code. It is composed of binary code, which is a sequence of 0s and 1s that the central processing unit (CPU) of the computer can immediately handle. Each instruction in machine level language relates to a specific hardware action and is unique to the architecture of the computer. Although machine code is very effective for computers, because it is binary in nature, it is difficult and time-consuming for humans to create and comprehend.

High-level languages offer a greater degree of abstraction and are created with readability for humans in mind. Examples of these languages are Python, Java, C++, and many more. Programmers may create code in these languages with familiar syntax and logical frameworks, which facilitates the conceptualization of algorithms and the resolution of challenging issues. Because high-level languages are platform-independent, the same code can operate with little to no changes on several computer types. They simplify chores and boost productivity with a plethora of built-in functions and libraries.

Compilation v/s Interpretation

Compilation and interpretation are the two main methods used in computer programming to convert source code into executable programs. These techniques each stand for different procedures, each with special benefits and drawbacks. Programmers must comprehend the distinctions between compilation and interpretation as they affect the way code is run and the functionality of software as a whole.

The process of **compilation** involves using a compiler to convert all of the source code into machine code or intermediate code. The compiler creates an executable file after thoroughly examining the source code and looking for syntax mistakes. The computer's hardware can directly execute the instructions in this file. Languages that are compiled, like C and C++, create executable files that may run independently of the original source code.

Speed is one of compilation's main benefits. The resultant executable file can be executed repeatedly without requiring recompilation after the code has been built. Because of this,

compiled languages are perfect for applications like system software, resource-intensive simulations, and video games where performance is crucial.

Even for big codebases, the compilation process alone might take a while. Furthermore, code that has been compiled is platform-specific, which means that it might not function on other operating systems or architectures without changes or recompilation.

On the other hand, **interpretation** entails running a program line by line and translating each one into either machine code or an intermediate code at runtime. Interpreters don't create separate executable files; instead, they read the source code, interpret it, and carry out the instructions. Python and JavaScript are two examples of interpreted languages that offer a great deal of flexibility and user-friendliness. They are well-liked for scripting and web development because they enable dynamic typing, interactive creation, and easier debugging.

Portability is one of the primary benefits of interpretation. Interpreted code may run on any system that has the matching interpreter installed since it is executed by a platform-specific interpreter. The distribution and deployment of programs are made easier by this portability, particularly in web-based contexts.

Conclusion

To sum up, learning programming opens doors to a world of limitless creativity and problem-solving. Gaining a solid understanding of programming principles enables people to drive technical improvements, create creative solutions, and influence the digital world. Being able to translate thoughts into reality is what makes learning a programming language so useful in today's technologically advanced society.

Learning programming promotes an attitude of constant learning and flexibility in addition to logical reasoning and analytical abilities. The groundwork established during the programming introduction serves as a launchpad for investigating more complex ideas, encouraging creativity, and forming a future in which the potential for accomplishment is boundless.

→SECTION 2—INTRODUCTION TO PYTHON

Among the many programming languages available, Python stands out for its ease of use, adaptability, and creativity. Since its introduction by Guido van Rossum in the late 1980s, Python has grown to become one of the most well-liked and extensively utilized programming languages globally. Its graceful syntax, simplicity of learning, and extensive ecosystem, which enable developers to take on a wide range of tasks, are responsible for its ascent to popularity.

Guido van Rossum developed the high-level programming language Python in the late 1980s. December 1989 marked the official start of its development, while February 1991 saw the release of Python 0.9.0. With a tribute to van Rossum's sense of humour, the language's name was inspired by the British comedy group Monty Python.

The design philosophy of Python places a strong emphasis on the readability and simplicity of the code, with the goal of offering a simple and clear means of expressing ideas. Python's popularity has grown quickly because of its simplicity and adaptability.

Python's primary characteristic is its simplicity. With just a few lines of code, programmers may express complicated concepts because of its clear and accessible syntax. Python is a great option for both novice and seasoned developers since it places a strong emphasis on readability and lowers the expense of program maintenance. Because there aren't any complicated symbols or language constructions, studying programming is easier and beginners may concentrate on understanding the principles rather than struggling with the syntax.

Python's popularity can be attributed in large part to its adaptability. Because it supports both procedural and object-oriented programming, this multiparadigm language may be tailored to fit a variety of development methods. Python is used in many different domains, including web development, data analysis, scientific computing, automation, and artificial intelligence. Its powers are increased by libraries like Django, Flask, Pandas, NumPy, and TensorFlow, which give developers strong tools to do a variety of jobs quickly.

Python is a great option for teaching programming topics at educational institutions because of its ease of use and adaptability. Because of its intuitive nature, teachers may concentrate on underlying ideas and make sure pupils understand them before moving on to more complicated subjects. Young brains are nurtured by Python's readability and instant feedback, which helps them develop into skilled programmers and problem solvers.

To sum up, the advent of Python in the programming world has caused a revolution that has made technology more accessible to everybody and encouraged a community of inventive and creative individuals. Both novices and professionals use it because of its ease of use, adaptability, and collaborative attitude. Python's continuing influence on learning, creativity, and problem-solving reinforces its status as a cornerstone in the ever-expanding world of programming languages, paving the way for a day when our imaginations are the only limits to what can be accomplished.

→SECTION 3—DOWNLOADING AND INSTALLING PYTHON

Learning to code has become increasingly important in the digital era, and Python has become a popular choice for both novice and seasoned developers due to its ease of use and adaptability. Installing and downloading the Python interpreter is the first step towards

starting the fascinating journey of Python programming. Because of how simple this procedure is, anybody with a computer and an internet connection can learn Python.

The first stop on the voyage is the official Python website, python.org, where you can easily download the most recent version of the language. Python is compatible with the majority of computers and comes in versions for Windows, macOS, and Linux, among other operating systems. With just a few clicks, users may choose the version that best suits their CPU architecture and operating system, simplifying and expediting the download process.

As soon as the Python installer is downloaded, the installation process begins. Python installation wizards facilitate the setup process for users by providing clear instructions and customizable options. After installation, users may quickly utilize Python from the command line by adding it to their system's PATH environment variable. This crucial step allows users to run Python programs and apps without having to manually find the installation location.

Although Python may be used straight from the command line, many developers find that their programming experience is improved when they use code editors or Integrated Development Environments (IDEs). IDLE, the built-in IDE for Python, PyCharm, and Visual Studio Code are popular options. These tools simplify the coding process and increase productivity by offering features like syntax highlighting, code completion, and debugging capabilities.

Finally, installing and downloading Python is the first step on a rewarding journey into the world of programming. With its large ecosystem of libraries and resources and its simple installation procedure, Python encourages novice programmers to develop their creativity, take on practical problems, and advance technical innovation. By making Python available to everyone, the programming community keeps expanding and creating a cooperative atmosphere where skills are refined, knowledge is exchanged, and creative solutions are developed. Installing and downloading Python may open doors to an infinite world of possibilities, regardless of your level of experience with coding.

→MODULE TEST 1

The aforementioned subjects are of utmost significance in the field of computer science and technology. In the current digital era, knowing the foundations of programming is not just a talent but also a must. Learning a programming language gives people the ability to be creative, figure out difficult issues, and make significant contributions to the development of technology. Python is particularly significant because of its ease of use and adaptability. Beginners will find it easy to understand basic programming principles because to its easy-to-read syntax and easy syntax. Furthermore, Python's extensive use in domains like data analysis, scientific computing, web development, and artificial

intelligence highlight its applicability across a range of industries. Furthermore, the first step to real-world application is learning how to download and install Python. This knowledge creates a new generation of tech-savvy workers by giving them the means to implement their programming ideas. All things considered, these subjects serve as the cornerstone upon which future programmers construct their knowledge, allowing them to confidently and skilfully traverse the always changing terrain of technology.

MODULE 2

PYTHON DATATYPES, VARIABLES, OPERATORS AND BASIC I/O OPERATIONS

Thus far, we have examined the foundational ideas of computer programming and the rich realm of Python programming. We studied the value of being proficient in programming in the technologically advanced world of today, with a focus on how programming encourages creativity and problem-solving skills. We also talked on the elegance and usefulness of Python, a programming language that is well-liked by both novice and experienced developers due to its ease of use and adaptability. We also looked at the necessary procedures for downloading and setting up Python, giving us the hands-on experience needed to get started with coding.

Equipped with this fundamental comprehension, we are now prepared to explore a new area in greater detail: PYTHON DATATYPES, VARIABLES, OPERATORS AND BASIC I/O OPERATIONS

→Section 1-- The “Hello, World!” Program

For those new to programming, the classic "Hello, World!" program in Python is a great place to start. Its importance is found in its simplicity. Python beginners may write their first interactive program with just one line of code, `print('Hello, World!')`. When this command is run, the computer is told to show "Hello, World!" on the screen, signalling the beginning of the coding adventure. This program, however simple, illustrates the 'print' function—which writes text to the console—and the basic organization of a Python script. Learning to use this basic program provides a strong foundation for newcomers, boosting their confidence and igniting their interest about the world of Python programming.

Let's see the how to write a simple “Hello, World!” Program in python:

`print(“Hello, World!”)`

This statement will print the statement-“Hello, World!” on the console.

The print statement is a built-in Python function that lets you show data on the terminal or console. Python evaluates text or variables enclosed in parenthesis and sends the result to the console when you use the print command. Although there are many intricate details involved, including interactions with the standard output stream, the procedure may be distilled down to a few crucial steps:

Function Invocation: The print function is called when you write `print("Hello, World!")` in a Python program. The text you wish to display is the argument, which is enclosed in parenthesis.

Argument Evaluation: The parameters in the print statement are assessed by Python. Strings, variables, expressions, and unique formatting sequences can all fall under this category.

Concatenation and formatting: If arguments are separated by commas, Python concatenates them into a single string before displaying them. If there are special formatting sequences (like `%s` or `{}`), Python produces the output in accordance with the format specified.

sending Processed Output to Standard Output Stream: Typically, you run your Python script on a console or terminal, which gets the output that has been processed. In most cases, this stream is connected to the console window.

Display on the Console: The formatted output allows you to see the results of the print statement on the console.

→SECTION 2—PYTHON LITERALS

Python is a ray of simplicity and adaptability in the huge field of programming, appealing to both novices and experienced programmers. One of the key ideas that helps us along the way as we learn more about Python is the idea of literals. Literals are the unaltered, raw values that we use in Python to define data in the programming world. They serve as the fundamentals from which we develop intricate algorithms and significant applications. Comprehending Python literals is similar to understanding the language's fundamental symbols, enabling us to have expressive and precise computer-to-human communication.

Python is a language that attracts both experienced developers and novices to its broad programming environment because of its simplicity and adaptability. The idea of literals is one of the key ideas that clears our way as we learn more about Python. Literals are the unaltered, raw values that we utilize to define data in Python when we are programming. They serve as the fundamental units around which intricate algorithms and significant applications are built. Knowing Python literals is like knowing the language's fundamental symbols; it enables us to have expressive and accurate conversations with the computer.

String literals are one of the several kinds of literals that are specifically used in Python programming. Python lets us build strings, which are character sequences, using single, double, or triple quotes. We can deal with textual material in a way that best meets our needs because of this flexibility. String manipulation, concatenation, substring extraction, and other operations enable us to write complex stories in our code. String literals are dynamic entities that can be changed and modified, making them more than just static text blocks. This allows us to create interactive and captivating user experiences.

Example:- `x = "Hello"`

Python's **numeric literals** support both floating-point and integer numbers, making it simple to carry out complex mathematical operations. By supporting complex numbers as well, Python's numeric literals enable us to go deeper into the fields of advanced mathematics and scientific computing. Numerical literals have a significant influence on our programming attempts because they enable us to develop algorithms that imitate real-world occurrences, compute financial forecasts, and model complicated systems.

Example:- `x=1`

`x=3.14`

Python literals are essentially the building blocks that we use to build our programs. Their accuracy and expressiveness enable humans to convey complex concepts to the computer, directing it through a series of actions that produce significant results. The capacity to convert intangible ideas into concrete solutions comes from mastering literals. When it comes to managing large datasets, creating sophisticated user interfaces, or modelling complex scientific models, Python literals are our constant allies, providing light on the route of our programming knowledge.

```
1 a=10
2 b=3.14
3 c="Hello"
4 print(type(a),type(b),type(c))
5
```

<class 'int'> <class 'float'> <class 'str'>

→SECTION 3—OPERATORS-DATA MANIPULATION TOOLS

Operators are multifaceted tools in the large field of programming that enable us to efficiently and precisely handle data and carry out a variety of computations. Operators are the foundation of data manipulation in Python, a dynamically typed and expressive language that enables us to convert unprocessed data into insightful understandings. We go on a trip where mathematical expressions, logical comparisons, and data transformations become our artistic canvas as we dig into the nuances of these operators. A sophisticated grasp of Python's operators empowers us to manoeuvre through intricate data structures, make wise choices, and create innovative and problem-solving algorithms.

Python's arithmetic operators let us carry out basic mathematical operations. These operators enable us to perform a wide range of mathematical operations, including addition, subtraction, multiplication, division, and remainder calculation. We may convert numeric literals and variables into dynamic entities using expressions like +, -, *, /, and %. This allows us to do computations ranging from simple math operations to complex scientific algorithms. We can make financial forecasts, simulate real-world scenarios, and analyse large datasets by becoming proficient with arithmetic operators.

Example:- a,b=10,20

c= a+b

c=30

Python's comparison operators enable us to understand the connections between values. Operators such as ==, !=, <, >, <=, and >= are utilized to assess conditions and perform logical operations. These operators are essential to decision-making processes because they allow us to regulate the flow of our programs and construct branching logic. We may construct sorting algorithms, search mechanisms, and validation checks by gaining insights into the relative magnitudes of data points through the comparison of variables and literals.

Example:- if a==b:

Print("YES")

Compares the value of a with b, if it matches, then Yes gets printed

In Python, logical operators serve as the fundamental units for making decisions. Operators like and, or, and not allow us to mix and work with logical assertions. With the help of these operators, we may construct complicated conditions that turn straightforward comparisons into devious decision trees. We build strong error-handling systems, develop algorithms with numerous conditions, and check user inputs through the use of logical operators. To guarantee the dependability and precision of our programs, we incorporate logical operators into our code.

Example:- if a==1 and b==2:

Print("Yes")

If both the conditions are satisfied, then only print statement will be executed.

In Python, bitwise operators explore the domain of binary representation. By utilizing operators like &, |, ^, ~, and <>>, we may manipulate discrete integer parts and solve the puzzles associated with binary data. Although bitwise operations may appear arcane, they are useful in data compression, encryption, and low-level programming. Understanding bitwise operators helps us to build complex data structures and optimize algorithms by providing us with insights into the binary underpinnings of digital systems.

Example:-a,b=10,5

c=a&b

Print(c)

Here, both 10(1010) and 5(0101) will be converted to binary numbers and on each bit, bitwise and operation is performed for which we will be getting a value of 0(0000).

```
a=10
b=20
if a<=b:
    print(a+b)
    print(a&b)
else:
    print("No")
```

```
30
0
> |
```

→SECTION 4: VARIABLES

Variables are strong elements in the complex web of programming languages that influence how we work with information, concepts, and solutions. Variables work as dynamic conduits in the world of Python, a language renowned for its ease of use and adaptability, connecting abstract ideas to concrete realities. We examine the fundamentals of variables via the prism of Python, learning how they enable us to store, manage, and modify data. Our systems come to life thanks to these dynamic placeholders, which also let us simulate intricate real-world events, store user inputs, and develop algorithms. We will travel to a place where creativity meets programming as we explore the subtleties of variables in Python, giving abstract concepts a tangible form.

Variables are names for values that have symbolic meaning in the computer language. These values can represent everything from basic integers to intricate arrays; they can also be texts, numbers, or complicated data structures. A symbolic connection is established between a variable's name and the data that has been saved when a value is assigned to it. Python's expressive ability lies in its symbolism, which enables us to handle a wide range of information with ease. We store not just text and numbers, but also concepts, goals, and the spirit of our original thought processes in variables.

The rules to follow for variables are:-

- Variable names ensure flexibility in naming standards by allowing letters (both uppercase and lowercase), numbers, and underscores, but not starting with a digit.
- Variable names with descriptive and meaningful names, such `total_count` or `user_input`, improve code readability and make the variable's function obvious.
- To avoid utilizing reserved terms in the language, don't name variables using Python keywords like `if`, `while`, or `print`.
- Snake_case, which separates words with underscores to improve consistency and readability, should be used for variables containing several words (e.g., `user_name`, `total_count`).
- Constants are program-wide variables that are always the same. They are usually identified by their names, which are in uppercase letters with underscores (e.g., `PI = 3.14159`).

- In order to minimize misunderstanding, ambiguous abbreviations should be avoided and variable names should be clear yet succinct.

Python variables can have several scopes that define where they can be edited or accessed inside a program. Global variables are accessible throughout the program, while local variables are restricted to particular functions or code blocks. Comprehending variable scope is essential for effective data management, averting inadvertent changes, and enhancing program efficiency. We can regulate how our data is contained and expanded by grasping variable scope, which guarantees that variables accomplish their intended goals without having unanticipated consequences.

Variables in Python are more than just placeholders; they comprise the essence of our programs, including concepts, principles, and goals. By utilizing the symbolism of variables, we are able to convert abstract ideas into concrete realities and depict the complexities of the environment we live in. We handle the complexity of data manipulation with grace and clarity thanks to Python's dynamic typing, variable name rules, and the idea of variable scope. We combine creativity with code when we work with variables, producing programs that are not only useful but also elegant representations of our creative intelligence. Variables in Python represent our capacity to turn ideas into reality in the dynamic field of programming, and as such, they are essential components of our technological innovation path.

→SECTION 5—COMMENTS

Within programming, comments serve as silent narrators, leading developers through the complexities of the code they produce. Lines of code morph into complex algorithms and creative solutions. Comments are useful annotations in Python's beautiful syntax; they give other programmers and future selves context, explanations, and insights. These short annotations are crucial for improving the readability, maintainability, and collaborative potential of Python code, even though they are frequently ignored.

Depending on the intricacy of the information they represent, comments in Python can be single- or multi-line comments. The hash symbol (#) indicates a single-line comment, which is succinct and appropriate for short explanations. For instance, the Python single-line comment # clearly states what the code after it is meant to do. Triple quotes (" or """) are used to create multi-line comments, which enable developers a larger canvas on which to write comprehensive explanations, describe complete functions, or draw boundaries between different areas of the software. Python writers may customize comments to meet the unique requirements of their code by striking the ideal balance between brevity and detail thanks to the flexibility offered by the comment kinds.

In collaborative programming projects, comments are essential because they serve as a conduit for communication among team members. When code is shared—inside a team or in open-source communities—comments clarify the reasoning and purpose of the code,

making it easier for others to understand, adapt, and expand upon pre-existing solutions. Furthermore, comments serve as lifelines for developers in the future—including the original programmer—who might need to go back and make changes to the code weeks, months, or even years after it was first written. They provide us a road map, so we can go back and examine our reasoning, comprehend the intent behind the code, and decide what needs to be fixed or debugged.

Python comments are more than just annotations; they are the threads that bind the programming community's collective understanding together. Their presence turns code into stories, making the complicated world of programming understandable, interesting, and accessible to everybody. As we learn to appreciate the art of Python commenting, we see how it may promote effective communication, teamwork, and code longevity. The subtle elegance of comments serves as a reminder that programming is about more than simply writing algorithms; it's also about exchanging information, mentoring other programmers, and pushing the frontiers of technology as a whole.

Example: # This is a Python comment that is one line long.

We designate comments using the hash symbol (#).

Note that the Python interpreter does not run comments.

Print("Hello, World!") # This comment clarifies the print statement's intent.

""" This Python comment consists of many lines.

To generate multi-line comments, use single or double triple quotations.

Multi-line comments are frequently used for docstrings or in-depth explanations.

“”

```
1 '''This is
2 used to comment
3 multiple lines'''
4 a=10
5 #This is used to comment a single line
6 b=20
```

→SECTION 6: INTERACTION WITH THE USER

User interaction in programming is the point where technology and human experience converge to create dynamic, captivating applications out of lines of code. Python is a language that is highly praised for its ease of use and adaptability. It is particularly good at

enabling smooth user interaction. Python's wide range of input and output functions enable programmers to design applications that engage users with personalized experiences and complicated algorithm execution, all while offering straightforward interfaces.

Python has a strong user input system that allows users to communicate with the program in real time. A mainstay of interactive Python programming, the `input()` method allows users to feed data directly into the program. Developers build interactive apps where users actively contribute to the program's flow by posing particular questions to users and using their answers as input. In addition to making computing more personalized, this user-driven method enables developers to create programs that react dynamically to user input, giving users a feeling of engagement and control.

Engaging with users entails more than just taking in information; in order to guarantee precision and dependability, it must be validated and processed. To check user input, Python developers utilize a variety of strategies, including conditional expressions and loops. Developers ensure that the software runs as intended by preventing unexpected inputs through the implementation of checks and restrictions. Python programs are made more reliable and user-friendly by validation techniques, which also reduce mistakes and direct users toward the right inputs, improving the user experience all around.

In Python, user interaction goes beyond input and includes giving users meaningful, contextually appropriate output. Python uses the `print()` method to interact with people, whether it's for reporting, presenting findings, or providing feedback. Developers may design messages that are understandable and straightforward by formatting the output and adding textual components and variables. Python programs become more informative by creating outputs that are easy to utilize; this helps users navigate complicated processes and provides them with insights that can be put to use.

Beyond simple command-line interfaces, Python provides robust frameworks for creating Graphical User Interfaces (GUIs), such as Tkinter, PyQt, and Kivy. GUIs improve user involvement by offering aesthetically pleasing interfaces with interactive components, buttons, and menus. With event-driven programming and drag-and-drop features, programmers create user-friendly apps that are both aesthetically beautiful and useful. With smooth navigation and an engaging visual experience, graphical user interfaces (GUIs) revolutionize the way users interact with software.

Python user interaction goes beyond simple data interchange; it's the human aspect in coding. Through gaining insight into the requirements, inclinations, and actions of users, developers may design programs that personally connect with consumers. Python's user interaction features let programmers create software that can react, listen, and change with the user, creating a sense of understanding and connection. Python enables developers to create apps that not only carry out tasks but also create deep, human-centred connections, enhancing the digital environment with intuitive, captivating, and user-friendly experiences. This is the synergy between technology and people.

Python user interaction is more than just data interchange; it's the human side of coding. Developers may design programs that have a personal connection with users by getting to know their requirements, interests, and habits. Python's user interface features let programmers create software that can interact with people by listening, responding, and changing with them to create a sense of understanding and connection. Python enables developers to create apps that generate meaningful, human-centric interactions in addition to carrying out tasks, enhancing the digital environment with intuitive, captivating, and user-friendly experiences. This is the synergy between technology and people that Python fosters.

Example: # Interaction with User: Asking for their name and greeting them
user_name = input("Enter your name: ") # Asks the user to input their name
print("Hello, " + user_name + "! Welcome to the Python World!") # Greet the user
with their inputted name.

→MODULE TEST 2

It had a series of questions based on basic topics which covered the topics which we covered in the section 2 of our course.

A simple "Hello, World!" program is frequently the starting point of a trip through the wide world of programming. This seemingly straightforward statement marks the beginning of a programmer's journey and the first instance of human purpose and machine execution. Writing the "Hello, World!" program in Python is a kind of rite of passage, the point at which a beginner becomes a programmer and begins a lifetime conversation with code.

Python programmers quickly go from simple text exchanges to the domain of user interaction. Developers may encourage users to participate by using the input() method, which transforms viewers from passive observers into active participants. Python applications become responsive, dynamic entities that adjust to individual preferences and choices by asking users for input and evaluating their answers.

One recurring theme in coding is the human factor. Beyond the technical aspects, developers explore the subtleties of human-computer interaction and the psychology of user behaviour. Python turns becomes a medium for empathy and comprehension, where programs react to human intents and subtle indications in addition to commands. Developers build an environment where technology becomes an extension of the user's intents by designing software that learns, connects, and adjusts. This fosters a sense of resonance and connection.

Python is the canvas and the brush in the vast tapestry of user interface, from the first "Hello, World!" to the complexities of GUIs and compassionate applications. It gives creators the ability to mold experiences, turning theoretical concepts into concrete,

interactive worlds. As we get to the end of our voyage, we see that user interaction with Python is an art form where meaningful connections arise and the lines between technology and users are blurred. This deep conversation is facilitated by Python's depth and simplicity, and it ushers in a new era in which human-computer connection is no longer limited by the limitations of code and instead becomes a beautiful symphony of purpose, expression, and comprehension.

MODULE 3

BOOLEAN VALUES, CONDITIONAL EXECUTION, LOOPS, LIST AND LIST PROCESSING, LOGICAL AND BITWISE OPERATIONS

So far, our investigation into Python has led us through the fundamental components of programming. Our modest introduction to the world of coding was the classic "Hello, World!" application. We explored the art of user interaction and how Python enables developers to build dynamic, responsive apps, going beyond simple output. By guaranteeing the precision and dependability of user inputs, we improved our data validation abilities. We now take a deeper exploration of the core functionality of Python, covering ideas such as Boolean values, loops, list structures, conditional execution, and logical and bitwise operations.

Python exposes us to the basic idea of true and false conditions in the context of Boolean values. We investigate conditional execution, which enables us to construct decision-making processes by directing a program's flow according to certain criteria. Another essential idea is loops, which allow us to perform tasks repeatedly, improving the automation and efficiency of our systems. List structures in Python provide flexible data manipulation capabilities that let us handle and store collections of things with ease. As we explore bitwise and logical operations, we learn about sophisticated computational methods and how Python manages intricate binary comparisons and manipulations.

These subjects serve as the fundamentals of complex Python programming, allowing us to formulate algorithms, work through issues, and design programs that react sensibly to various inputs and circumstances. We get a deeper comprehension of Python's possibilities via this investigation, opening the door to more complex and nuanced coding experiences.

→SECTION 1—MAKING DECISIONS IN PYTHON

Making decisions is a fundamental component of programming, directing the course of a program according to particular parameters and requirements. Python's sophisticated syntax and strong structures enable developers to make decisions that shape the behaviour of their applications by guiding them through the complex world of logic. As we explore the core ideas that enable programmers to develop dynamic, responsive, and intelligent software, we also explore the complexities of conditional statements and the art of decision-making in Python.

Conditional statements are the foundation of Python decision-making and are what allow for dynamic program execution. Logic criteria that dictate a program's next step may be defined by developers using expressions like `if`, `elif` (short for "else if"), and `else`. Programmers can construct decision trees using these conditional statements, in which distinct code branches are carried out in response to the assessment of particular criteria. Python programs become context-aware and adaptable by use conditional expressions to respond dynamically to a variety of contexts.

Understanding Boolean values, `True` and `False`, the basic building blocks of logical statements, is the first step towards making judgments in Python. Developers can assess whether conditions are true or false by using boolean values as the foundation for comparisons. Python programmers construct logical expressions that serve as the foundation for decision-making processes using relational operators (`==` for equality, `!=` for inequality, `<` for less than, `>` for greater than, `<=` for less than or equal to, and `>=` for greater than or equal to). The software may compare variables, user inputs, or computed values thanks to these expressions, and it can make judgments depending on the conclusions drawn from these comparisons.

Python decision-making also applies to loops, wherein some code blocks are repeated based on predetermined circumstances. Developers may establish iterative decision-making processes by using loop structures like `while` and `for`, which enable programs to do tasks more than once depending on predetermined circumstances. With the help of loops, developers may manage data processing in a dynamic manner and take judgments and actions on large datasets or collections of things.

EXAMPLE:# Decision-Making Example: Checking Eligibility to Vote

Asking the user to input their age

```
user_age = int(input("Please enter your age: "))
```

Checking if the user is eligible to vote

```
voting_age = 18 # Minimum age required to vote in many countries
```

```
if user_age >= voting_age:
```

```
    print("Congratulations! You are eligible to vote.")
```

```
else:
```

```
years_to_wait = voting_age - user_age
print(f"Sorry, you are not eligible to vote yet. You need to wait {years_to_wait}
years.")
```

```
1 a=int(input())
2 if a>=18:
3     print("OKAY Eligible")
4 else:
5     print("Not Eligible")
6 |
```

input

```
17
Not Eligible
```

→SECTION 2—LOOPS IN PYTHON

Automation and efficiency are critical in the field of programming. Here's where Python loops become indispensable tools, leading developers elegantly and precisely along the road of iteration. Programmers may carry out tasks on datasets, collections, or sequences of items quickly by using loops, which let them run a set of instructions repeatedly. As we explore the world of Python loops, we discover the strength and adaptability of these structures, realizing how they improve productivity, simplify procedures, and enable developers to skilfully solve challenging issues.

- The core of Python's looping functionality is the **for** loop, a flexible design that makes iterating through sequences easier. The for loop iterates over each element in lists, strings, and other iterable objects with ease. The loop is perfect for jobs that need a predefined number of iterations since it can be used with methods like `range()` to allow developers to iterate a certain number of times. The for loop is an excellent tool for processing data because it makes it simple for programmers to modify, examine, and alter data, which improves the overall effectiveness of their applications.

Example: # Example of a for loop iterating through a list of numbers

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

Output: 1, 2, 3, 4, 5 (each number on a new line)

```
1- for i in range(1,10):
2-     print(i,end=" ")
3-
input
1 2 3 4 5 6 7 8 9
```

- The **while** loop provides dynamic iteration based on predetermined circumstances, whereas the for loop is great for iterating over sequences. As long as the specified condition is true, the loop keeps running. This adaptability offers flexibility and adaptability in a range of situations by enabling developers to handle challenges where the number of repetitions is unknown or unpredictable. To avoid endless loops, it is imperative to provide a mechanism that will finally end the repetition.

Example: # Example of a while loop iterating until a condition is met

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Output: 1, 2, 3, 4, 5 (each number on a new line)

```
1 count=1
2 while count<=5:
3     print(count)
4     count+=1
5
1
2
3
4
5
```

- The loop control statements—break, continue, and else—further enhance Python loops. The break statement gives programmers a way out of a loop early in the event that a certain condition is satisfied. Conversely, selective processing is made possible by the continue statement, which bypasses the remaining code in the loop and jumps to the following iteration. Furthermore, the loop-related else block is run at the loop's completion of its iterations in the absence of a break statement. The iterative process is made more precise and sophisticated by these control instructions, which enable programmers to fine-tune the behaviour of their loops.

Example: for i in range(10):

```
    if i==5:
        print("YES")
        break #It will come out of the loop
```



```
1 for i in range(10):
2     if i==5:
3         print("YES")
4         break
5     else:
6         print(i)
7
```

```
0
1
2
3
4
YES
```

Python loops are more than just structural elements; they are the engine of automation, allowing programmers to carry out tedious operations systematically and effectively. Loops are useful for processing large datasets, automating computations, and solving complicated issues because they offer a foundation upon which more sophisticated solutions may be constructed. Python's loop constructions and its control statements are incredibly elegant, allowing developers to confidently walk the iterative route, reshaping programs and turning algorithms into dynamic, efficient processes. Python loops, with their strength and simplicity, capture the spirit of programming: the art of automation, which turns tedious jobs into elegant, efficient solutions that highlight computational expertise and human creativity.

→SECTION3—LOGIC AND BIT OPERATIONS IN PYTHON

Logic and bit operations are strong tools in the complex world of programming that programmers work with data at a fundamental level. Python's flexible syntax and wide range of logical and bitwise operators enable programmers to create sophisticated algorithms, carry out challenging comparisons, and enhance data processing and storage. We dissect the intricacies of bit and logic operations in Python as we investigate these domains, comprehending their importance and practical uses in real-world programming situations.

Three basic **logical operators** are available in Python: and, or, and not. With the use of these operators, programmers may construct compound conditions, which combine many expressions to evaluate a statement's truth or falsehood. Only when both of the criteria it links to are true does the and operator evaluate to True. On the other hand, if at least one of the related criteria is true, the or operator evaluates to True. The unary operator not negates an expression's truth value by changing True to False and vice versa. Complex conditional statements and decision-making procedures are constructed using these logical operators as building blocks.

Example: x,y=5,10

```
# Using logical operators to create compound conditions
if x > 0 and y > 0:
```

```
print("Both x and y are positive.")    #TRUE
```

```
if x > 0 or y < 0:  
    print("Either x is positive or y is negative.")    #TRUE
```

```
if not x < 0:  
    print("x is not negative.")    #TRUE
```

```
1  # Example of logical operators in Python  
2  x = 5  
3  y = 10  
4  
5  # Using logical operators to create compound conditions  
6  if x > 0 and y > 0:  
7      print("Both x and y are positive.")  
8  
9  if x > 0 or y < 0:  
10     print("Either x is positive or y is negative.")  
11  
12  if not x < 0:  
13     print("x is not negative.")  
14  
15  
16
```

```
Both x and y are positive.  
Either x is positive or y is negative.  
x is not negative.
```

Python's bitwise operators enable programmers to work with individual data bits by carrying out binary operations. & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), and >> (right shift) are some examples of bitwise operators. Applications for these operators include data compression, encryption, storage optimization, and low-level hardware interface. Developers may construct effective algorithms for tasks like image processing, network communication, and data encryption by utilizing bitwise operations.

Example: # Example of bitwise operators in Python

a = 5 # Binary: 0101

b = 3 # Binary: 0011

Bitwise AND

result_and = a & b # Result: 0001 (Decimal: 1)

Bitwise OR

result_or = a | b # Result: 0111 (Decimal: 7)

Bitwise XOR

result_xor = a ^ b # Result: 0110 (Decimal: 6)

Bitwise NOT

result_not_a = ~a # Result: 1111010 (Decimal: -6)

Left Shift (a << 2)

left_shifted = a << 2 # Result: 10100 (Decimal: 20)

Right Shift (a >> 1)

right_shifted = a >> 1 # Result: 10 (Decimal: 2)

print("Bitwise AND:", result_and)

print("Bitwise OR:", result_or)

print("Bitwise XOR:", result_xor)

print("Bitwise NOT (a):", result_not_a)

print("Left Shift (a << 2):", left_shifted)

print("Right Shift (a >> 1):", right_shifted)

Output:

Bitwise AND: 1

Bitwise OR: 7

Bitwise XOR: 6

Bitwise NOT (a): -6

Left Shift (a << 2): 20

Right Shift (a >> 1): 2

```

1  # Example of bitwise operators in Python
2  a = 5  # Binary: 0101
3  b = 3  # Binary: 0011
4
5  # Bitwise AND
6  result_and = a & b  # Result: 0001 (Decimal: 1)
7
8  # Bitwise OR
9  result_or = a | b  # Result: 0111 (Decimal: 7)
10
11 # Bitwise XOR
12 result_xor = a ^ b  # Result: 0110 (Decimal: 6)
13
14 # Bitwise NOT
15 result_not_a = ~a  # Result: 11111010 (Decimal: -6)
16
17 # Left Shift (a << 2)
18 left_shifted = a << 2  # Result: 10100 (Decimal: 20)
19
20 # Right Shift (a >> 1)
21 right_shifted = a >> 1  # Result: 10 (Decimal: 2)
22
23 print("Bitwise AND:", result_and)
24 print("Bitwise OR:", result_or)
25 print("Bitwise XOR:", result_xor)
26 print("Bitwise NOT (a):", result_not_a)
27 print("Left Shift (a << 2):", left_shifted)
28 print("Right Shift (a >> 1):", right_shifted)
29

```

```

Bitwise AND: 1
Bitwise OR: 7
Bitwise XOR: 6
Bitwise NOT (a): -6
Left Shift (a << 2): 20
Right Shift (a >> 1): 2

```

Python's bit operations and logic are more than just computing tools; they form the foundation for sophisticated algorithms that let programmers handle challenging issues elegantly and effectively. Programmers who understand these operations will be able to navigate the world of truth, falsehood, and data optimization, paving the way for a day when bit manipulation and logic will be used to decode and simplify computational complexity.

→SECTION 4—LISTS

The capacity to arrange, work with, and store data is essential in the broad field of programming. The Python programming language, renowned for its ease of use and adaptability, provides a robust data structure called lists. Many programs are built on lists, which give programmers an adaptable and dynamic way to manage groups of elements. We enter a realm where data finds structure, complexity meets simplicity, and programmers use their skills to turn abstract ideas into concrete, well-organized information as we explore the nuances of lists in Python.

In Python, a list is fundamentally an ordered set of items, each uniquely identified by a key or index. Python lists are extremely adaptable since they may include components of multiple data types, unlike arrays in certain other programming languages. Python lists may hold a broad range of data, including numbers, strings, objects, and even other lists, enabling programmers to design intricate data structures and representations.

Example of creating a list in Python

```
fruits = ["apple", "banana", "orange", "grape", "kiwi"]
```

python

Copy code

```
# Example of creating a list in Python
fruits = ["apple", "banana", "orange", "grape", "kiwi"]
```

Lists in Python provide an abundance of operations that help programmers shape, work with, and explore their contents. Programmers may easily add, change, and retrieve elements from lists using operations like adding, inserting, slicing, and concatenating. The ability to nest lists within lists and the flexibility to resize lists dynamically provide up creative possibilities for representing complex hierarchical systems.

Example of list operations in Python

```
fruits.append("mango") # Adds "mango" to the end of the list
```

```
fruits.insert(1, "pear") # Inserts "pear" at index 1
```

```
print(fruits[2:5]) # Outputs elements from index 2 to 4: ['banana', 'orange', 'grape']
```

```
vegetables = ["carrot", "broccoli"]
```

```
combined_list = fruits + vegetables # Concatenates two lists
```

```
print(combined_list) # Outputs: ['apple', 'pear', 'banana', 'orange', 'grape', 'mango', 'carrot', 'broccoli']
```

python

Copy code

```
# Example of list operations in Python
fruits.append("mango") # Adds "mango" to the end of the list
fruits.insert(1, "pear") # Inserts "pear" at index 1
print(fruits[2:5]) # Outputs elements from index 2 to 4: ['banana', 'orange', 'grape']
vegetables = ["carrot", "broccoli"]
combined_list = fruits + vegetables # Concatenates two lists
print(combined_list) # Outputs: ['apple', 'pear', 'banana', 'orange', 'grape', 'mango', 'carrot', 'broccoli']
```

List comprehensions in Python provide data transformation a refined touch and let programmers generate lists in a clear and expressive way. Developers may create new lists by applying operations and conditions to preexisting ones, which facilitates the efficient filtering, mapping, and manipulation of data.

Example of list comprehension in Python+

```
numbers = [1, 2, 3, 4, 5]
```

```
squared_numbers = [num ** 2 for num in numbers] # Squares each number in the list
```

```
print(squared_numbers) # Outputs: [1, 4, 9, 16, 25]
```

Python lists are used in a variety of fields, including web development, artificial intelligence, data processing, and analysis. Lists are used in data science as containers for datasets, enabling analysts to create visualizations and carry out statistical computations. Lists in web development make it easier to organize user inputs and database queries so that everything works together smoothly. In the field of artificial intelligence, lists make it possible to construct intricate data structures like trees and graphs, which serve as the basis for very complicated algorithms.

Python lists are more than simply data containers; they are efficiency and organization personified. They provide an organized method for handling data, enabling programmers to convert unstructured data into insightful understandings. Because of lists' dynamic properties, flexible operations, and sophisticated comprehensions, programmers can solve complicated problems with accuracy and simplicity. The digital world may flourish on organized knowledge, inventiveness, and the endless possibilities that Python lists provide to the field of programming as developers use lists to build a tapestry of structured data.

→SECTION 5—SORTING SIMPLE LISTS : THE BUBBLE SORT ALGORITHM

Sorting is one of the core jobs in the large field of algorithms, converting disorganized input into ordered sequences. The Bubble Sort algorithm stands out among other sorting algorithms as a straightforward but sophisticated method. Because of its simple implementation and obvious approach, it is a fundamental idea in computer science. We dissect the Bubble Sort algorithm to discover the fundamentals of this age-old sorting technique, comprehending its ideas via practical applications.

The idea behind **bubble sort** is to constantly switch nearby components that are in the incorrect order. The method goes over the list more than once, exchanging and comparing neighbouring entries until the full list is sorted. The term "Bubble Sort" refers to the way that smaller items, or bubbles, rise to the top of the list and bigger items fall to the bottom, progressively creating the sorted order.

Let's consider a simple list of numbers: [5, 2, 9, 1, 5, 6]. Applying the Bubble Sort algorithm step by step:

First Pass:

Compare 5 and 2. Since 5 is greater, swap them: [2, 5, 9, 1, 5, 6].

Compare 5 and 9. No swap needed.

Compare 9 and 1. Swap them: [2, 5, 1, 9, 5, 6].

Compare 9 and 5. Swap them: [2, 5, 1, 5, 9, 6].

Compare 9 and 6. Swap them: [2, 5, 1, 5, 6, 9].

After the first pass, the largest element (9) is in its correct position at the end of the list.

Second Pass:

Compare 2 and 5. No swap needed.

Compare 5 and 1. Swap them: [2, 1, 5, 5, 6, 9].

Compare 5 and 5. No swap needed.

Compare 5 and 6. No swap needed.

After the second pass, the second largest element (6) is in its correct position.

Third Pass:

Compare 2 and 1. Swap them: [1, 2, 5, 5, 6, 9].

Compare 2 and 5. No swap needed.

After the third pass, the list is completely sorted.

Bubble Sort has limits even if its simplicity is one of its strengths. Due to its $O(n^2)$ time complexity, it is not as effective for handling huge datasets and is hence not as appropriate for real-time applications where speed is essential. Nonetheless, Bubble Sort functions as a fundamental idea, providing a concise overview of the fundamentals of sorting algorithms and clearing the path for comprehension of more complex methods like Quick Sort and Merge Sort.

Despite its simplicity, the Bubble Sort algorithm teaches important insights about algorithmic design and problem-solving. Its simplicity serves to highlight the basic operations of comparisons and swaps, which form the basis of more intricate sorting algorithms. In addition to learning a basic computer science topic, developers who investigate the beauty of Bubble Sort also have a better knowledge of the analytical and iterative process involved in developing algorithms. Bubble Sort serves as a timeless guide in the transition from chaos to order, teaching programmers that deep insights may be learned from even the most basic tasks, laying the groundwork for all of their computational pursuits.

→SECTION 6 – OPERATIONS ON LISTS

Python lists are flexible data structures that make it easy for programmers to store and manage collections of elements in the ever-evolving world of programming. With Python's abundance of operations, programmers may easily manipulate, arrange, and derive valuable information from lists. As we explore Python's operations on lists, we explain how these operations improve developers' productivity and creativity by revealing the breadth of their usefulness through practical examples.

Python has a number of methods for creating lists, including list comprehension, individual element specification, and list conversion from other data structures like strings or tuples. Lists can have components added, changed, or removed as required since they are dynamic after they are formed.

Example of list creation and modification in Python

```
fruits = ["apple", "banana", "orange", "grape"]
```

Adding a new element to the list

```
fruits.append("mango") # Result: ["apple", "banana", "orange", "grape",  
"mango"]
```

Modifying an element in the list

```
fruits[1] = "cherry" # Result: ["apple", "cherry", "orange", "grape",  
"mango"]
```

```
1 # Example of list creation and modification in Python
2 fruits = ["apple", "banana", "orange", "grape"]
3
4 # Adding a new element to the list
5 fruits.append("mango") # Result: ["apple", "banana", "orange", "grape", "mango"]
6
7 # Modifying an element in the list
8 fruits[1] = "cherry" # Result: ["apple", "cherry", "orange", "grape", "mango"]
9 print(fruits)
```

input

```
['apple', 'cherry', 'orange', 'grape', 'mango']
```

Python's list slicing feature makes it possible for programmers to quickly extract particular subsets from lists. Programmers can build sublists to help with activities like data analysis, modification, and visualization by providing start and end indices.

Example of list slicing in Python

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Extracting a sublist from index 2 to 5

```
sublist = numbers[2:6] # Result: [3, 4, 5, 6]
```

```
1 # Example of list slicing in Python
2 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
3
4 # Extracting a sublist from index 2 to 5
5 sublist = numbers[2:6] # Result: [3, 4, 5, 6]
6 print(sublist)
```

```
[3, 4, 5, 6]
```

Python programming's beauty is perfectly embodied in list comprehensions. By applying operations on preexisting lists, they let developers generate new ones and reduce the number of lines of code needed to write a single statement.

Example of list comprehension in Python

```
numbers = [1, 2, 3, 4, 5]
```


Squaring each number in the list using list comprehension

squared_numbers = [num ** 2 for num in numbers] # Result: [1, 4, 9, 16, 25]

```
1 # Example of List comprehension in Python
2 numbers = [1, 2, 3, 4, 5]
3
4 # Squaring each number in the List using List comprehension
5 squared_numbers = [num ** 2 for num in numbers] # Result: [1, 4, 9, 16, 25]
6 print(squared_numbers)
```

input

[1, 4, 9, 16, 25]

Lists may be sorted and reversed in Python using built-in methods that let programmers put components in either ascending or descending order and change the order of the list.

Example of sorting and reversing lists in Python

numbers = [5, 2, 9, 1, 3]

Sorting the list in ascending order

sorted_numbers = sorted(numbers) # Result: [1, 2, 3, 5, 9]

Reversing the list

reversed_numbers = list(reversed(numbers)) # Result: [3, 1, 9, 2, 5]

```
1 # Example of sorting and reversing Lists in Python
2 numbers = [5, 2, 9, 1, 3]
3
4 # Sorting the List in ascending order
5 sorted_numbers = sorted(numbers) # Result: [1, 2, 3, 5, 9]
6 print(sorted_numbers)
7 # Reversing the List
8 reversed_numbers = list(reversed(numbers)) # Result: [3, 1, 9, 2, 5]
9 print(reversed_numbers)
```

input

[1, 2, 3, 5, 9]
[3, 1, 9, 2, 5]

Python's operations on lists provide access to innovative problem-solving techniques. Developers may take on a variety of tasks, from algorithm creation and optimization to data analysis and manipulation, by being proficient in these procedures. Lists become more than simply data containers; they become dynamic entities that react to commands from the programmer, making it possible to turn abstract concepts into concrete, organized data.

To sum up, Python's operations on lists capture the spirit of effective and versatile computing. They provide smooth data organization, transformation, and analysis by giving programmers the ability to manoeuvre across the complicated data landscape. By utilizing these processes, developers may open up a world of possibilities, forming creative solutions and expanding the creative and functional possibilities of programming.

→SECTION 7 – LISTS IN ADVANCED APPLICATIONS

Python lists are essential for advanced programming because they provide a flexible and dynamic method of managing data. These modest structures, which are able to accommodate a group of components, have a wide range of complex applications, ranging from web development and gaming to data research and artificial intelligence. We discover the transforming potential of lists as we investigate their sophisticated uses; examples from everyday life highlight their inventiveness and applicability.

Lists are the foundation of **data science** and are used to manage massive databases. Lists are now necessary for storing and handling data points because to Python's extensive modules, such NumPy and Pandas. Data scientists may carry out complex operations, statistical analysis, and machine learning tasks using lists, opening the door to novel discoveries.

Example of using lists in data science with NumPy

import numpy as np

Creating a NumPy array from a list

data_list = [1, 2, 3, 4, 5]

numpy_array = np.array(data_list)

Performing operations on the NumPy array

mean_value = np.mean(numpy_array) # Calculate mean: 3.0

```
1
2 # Example of using lists in data science with NumPy
3 import numpy as np
4
5 # Creating a NumPy array from a list
6 data_list = [1, 2, 3, 4, 5]
7 numpy_array = np.array(data_list)
8
9 # Performing operations on the NumPy array
10 mean_value = np.mean(numpy_array) # Calculate mean: 3.0
11 print(mean_value)
```

3.0

Lists help with dynamic user interfaces and content management in **web development**. Databases, user inputs, and other APIs may all be queried and manipulated dynamically by developers thanks to lists, which hold data. Web apps provide dynamic interfaces and smooth user experiences by utilizing lists.

Example of using lists in web development

user_comments = ["Great work!", "I love this feature!", "Awesome website!"]

Displaying user comments dynamically on a webpage

for comment in user_comments:

```
print("<div>" + comment + "</div>")
```

```
1
2 # Example of using Lists in web development
3 user_comments = ["Great work!", "I love this feature!", "Awesome website!"]
4
5 # Displaying user comments dynamically on a webpage
6 for comment in user_comments:
7     print("<div>" + comment + "</div>")
8
```

input

```
<div>Great work!</div>
<div>I love this feature!</div>
<div>Awesome website!</div>
```

Lists are useful data structures for **pathfinding algorithms** in the gaming industry, which allow objects and characters to move around the game environments with intelligence. Lists enable game creators to integrate complex artificial intelligence and improve gameplay interactions by storing movement pathways, waypoints, and grid placements.

Example of using lists for pathfinding in gaming

```
waypoints = [(1, 2), (3, 4), (5, 6), (7, 8)]
```

Implementing pathfinding algorithm using waypoints stored in a list

```
def find_path(start, end, waypoints):
```

```
    # Implementation logic for pathfinding algorithm
```

```
    pass
```

Finding a path from (1, 2) to (7, 8) using the waypoints list

```
path = find_path((1, 2), (7, 8), waypoints)
```

Lists provide for real-time data display and analysis in **scientific applications**.

Experimental data points may be gathered and stored by scientists in lists, which can later be displayed with the help of programs like Matplotlib. Lists are essential for displaying data sets because they let researchers examine patterns and come to well-informed conclusions.

Example of using lists for real-time data visualization

```
import matplotlib.pyplot as plt
```

Experimental data points stored in lists

```
time_points = [1, 2, 3, 4, 5]
```

```
measurement_values = [10, 12, 8, 15, 11]
```

Creating a line chart to visualize the data

```
plt.plot(time_points, measurement_values)
```

```
plt.xlabel('Time')
```

```
plt.ylabel('Measurement Values')
```

```
plt.title('Real-time Data Visualization')
```

plt.show()

In sophisticated systems, lists are more than just data structures—they are forces behind advancement and creativity. In the hands of developers, their effectiveness and adaptability make them priceless tools that facilitate the development of dynamic solutions in a variety of fields. The use of lists in programming is changing as a result of technological advancements, influencing contemporary applications and raising the bar for what is possible. Lists are the unsung heroes of the digital world, enabling sophisticated applications and fostering the growth of a more intelligent, inventive, and connected digital environment.

→MODULE 3—COMPLETION TEST

We have taken a thorough investigation of the complexities of programming, moving from decision-making in Python to comprehending lists in sophisticated programs. These subjects, which range from basic control structures to sophisticated methods of data manipulation, are the cornerstones of computational problem solving.

Comprehending decision-making elements like conditionals and loops enables programmers to develop clever, reacting algorithms. By working with data structures like lists, we may learn how to dynamically alter, iterate over, and display data in addition to storing it. As we go, we come across algorithms such as Bubble Sort, which reveal the beauty of sorting methods essential to many different uses.

The subjects addressed stress the significance of efficiency and rationality in programming. The same principles apply whether making complex judgments in real-time applications, precisely iterating over data sets, or using sophisticated algorithms for optimization: efficiency, rationality, and clarity are crucial.

Furthermore, the exploration of sophisticated applications demonstrates the practical uses of programming. Proficiency in data science, scientific research, gaming, dynamic web development, and gaming goes beyond theoretical understanding. They provide programmers the ability to handle big databases, make sophisticated game tactics, construct interactive websites, and depict intricate scientific data.

This comprehensive view emphasizes that programming is about creating solutions, encouraging creativity, and solving problems in the real world—it's not simply about writing code. Every topic covered, from sophisticated data management to decision-making, adds to a more comprehensive understanding of computational problem-solving. As we get to the end of our investigation, we see that these subjects are stepping stones that promote lifelong learning, creativity, and a greater understanding of the art and science of programming.

MODULE 4

FUNCTIONS, TUPLES, DICTIONARIES, EXCEPTIONS AND DATA PROCESSING

Thus far, our exploration of Python programming has yielded a strong basis, encompassing fundamental ideas ranging from elementary syntax to sophisticated uses. We now explore functions, tuples, dictionaries, exceptions, and data processing techniques as we go into increasingly specialized fields. Functions provide code structure and reuse as building blocks of modular programming. Dictionary entries and tuples offer sophisticated methods for effectively managing and storing data. By enabling us to elegantly handle failures, exception handling strengthens the resilience of our applications. Furthermore, mastering data processing procedures improves our capacity for efficient information manipulation and analysis. Let's explore these subjects in more detail, revealing the subtleties of Python's powers in this dynamic world of programming.

→SECTION 1—FUNCTIONS

Functions are essential building blocks in programming that help to organize code more efficiently by encapsulating activities and encouraging reusability. Python provides developers with a powerful toolkit to efficiently write functions because to its clean syntax and readability. We dissect the core of modular programming—the process of breaking complicated problems down into smaller, more manageable components, each with a defined purpose—as we investigate the idea of functions in Python.

- When a function is declared, its name, return type, and any parameters are specified.
- A function must be called in order to be carried out. To do this, use the name of the function followed by parentheses holding the parameters.
- Variables specified inside a function are local to that function as functions have their own scope. Until they are specifically returned, they cannot be accessed from outside of it.
- The return statement is used by functions to return values. One can utilize the returned value directly or assign it to a variable. A function implicitly returns None in the absence of a return statement.
- Functions can accept parameters, also known as arguments, which serve as placeholders for the values that will be used when the function is called. Function calls use parameters, which are defined in the function declaration.
- Code reusability is one of functions' main benefits. Programming may be made modular and efficient by calling a function more than once with various parameters once it has been defined.
- The idea that functions can call themselves is called recursion. This permits elegant solutions to some issues, but in order to prevent infinite loops, an explicit termination condition is needed.

Fundamentally, a named chunk of code in Python that carries out a certain purpose is called a function. After processing the input parameters, it may or may not deliver a result. Functions improve the readability and organization of code, allowing programmers to create programs that are simple, clear, and manageable. This is an illustration of a basic function that determines a number's square:

Example of a function in Python

def square(number):

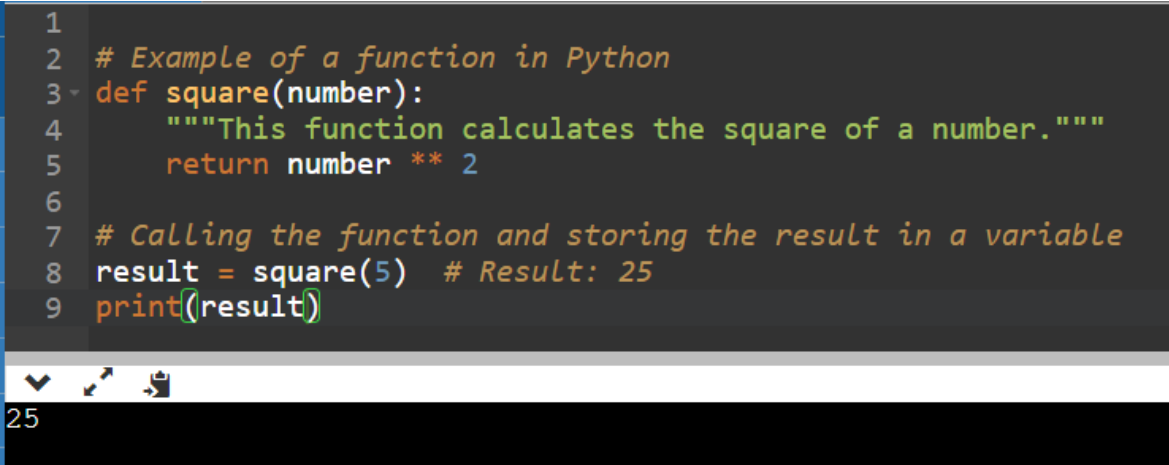
"""This function calculates the square of a number."""

return number ** 2

Calling the function and storing the result in a variable

result = square(5) # Result: 25

```
1
2 # Example of a function in Python
3 def square(number):
4     """This function calculates the square of a number."""
5     return number ** 2
6
7 # Calling the function and storing the result in a variable
8 result = square(5) # Result: 25
9 print(result)
```



→SECTION 2—HOW FUNCTIONS COMMUNICATE WITH THEIR ENVIRONMENT

Programming functions are established mechanisms that allow a program to communicate with its environment in addition to encapsulating code for certain tasks. For information, control, and data to move throughout a program, this interaction is essential. With appropriate examples, we shall examine how functions interact with their surroundings in this article.

- **Parameters and Arguments:** Parameters and arguments are two main ways that functions interact with their surroundings. While arguments are the actual values given during the function call, parameters serve as placeholders for values that a function expects.

Example: def greet(name):
 print("Hello, " + name + "!!")
 greet("John")

Output: Hello, John!

```
1
2 def greet(name):
3     print("Hello, " + name + "!")
4
5 greet("John")
```




- **Return Values:** Functions frequently use return values to convey results to the caller code. This enables functions to return data or results to the section of the program from where they were called.

Example: `def square(x):`
 `return x * x`

`result = square(5)`
`print(result)`

Output: 25

```
1
2 def square(x):
3     return x * x
4
5 result = square(5)
6 print(result)
7
```



- **Global and Local Scope:** The scope of a function determines the locations at which variables can be accessed. Unless specifically designated as global, variables declared inside a function are local to that function. In addition to facilitating clear communication between functions and their surroundings, this aids in the prevention of unwanted side effects.

Example: `global_var = 10`

`def multiply_by_global(x):`
 `return x * global_var`
`result = multiply_by_global(5)`
`print(result)`

Output: 50

```
1
2 global_var = 10
3
4 def multiply_by_global(x):
5     return x * global_var
6
7 result = multiply_by_global(5)
8 print(result)
9
```

50

To sum up, functions can interact with their surroundings through a variety of means, including parameters, return values, scope, side effects, and more. Writing efficient, modular, and maintainable code requires an understanding of these communication channels. Programmers may promote code clarity and reusability by utilizing these concepts to develop systems where functions interact with one another without any problems.

→SECTION 3—RETURNING A RESULT FROM A FUNCTION

Programming functions are fundamental building elements that let programmers encapsulate logic and produce reusable code segments. The capacity of functions to provide outcomes is one of the main characteristics that enhances their adaptability and usefulness. An essential idea that improves code structure, readability, and overall program efficiency is the practice of returning a result from a function.

Essence of Returning Values: Fundamentally, the act of a function providing a result allows the function to interact with other parts of the program. Returning a result enables the function to supply its computed or processed data to the code that called it, as opposed to completing a job and containing the result inside the function.

Syntax and Mechanism: The return keyword must be used in conjunction with the expression or variable whose value is meant to be returned to the caller code in the majority of programming languages in order to return a result from a function. Functions can be dynamic and sensitive to various inputs thanks to this approach.

Example: `def add_numbers(x, y):`

`result = x + y`

`return result`

`x=add_numbers(1,2)`

`print(x)`

Output: 3

```
1
2 def add_numbers(x, y):
3     result = x + y
4     return result
5 x=add_numbers(1,2)
6 print(x)
7
```

3

Facilitating Reusability: Code reusability is greatly enhanced by the ability to return results. The output of a function can be used in other parts of the program once it has completed a particular job or computation. Because functions become separate components that may be readily merged into other portions of the codebase, this encourages the use of modular programming techniques.

Supporting Decision Making: Functions are frequently involved in program decision-making procedures. Functions can transmit information that affects later actions by delivering results. This makes it possible to create more dynamic and flexible programs where the results of one function can influence the actions of other functions.

Error Handling and Special Cases: In order to handle incorrect circumstances or exceptional instances within a function, returning results is essential. Functions can convey problems to the caller code so that it can react correctly by returning certain values or by invoking exceptions.

Programming's core concept of producing a result from a function enables programmers to write modular, legible, and effective code. This feature allows code reuse and efficient handling of complicated processes in addition to facilitating communication between various computer components. Developers contribute to the production of reliable and maintainable software solutions as they learn to use the power of returning outcomes.

→SECTION 4—SCOPES IN PYTHON

The notion of scopes is essential to establishing the visibility and accessibility of variables in Python, a programming language renowned for its clarity and simplicity. Writing programs that are comprehensible, manageable, and free of errors requires a grasp of scopes, which specify where in the code a certain variable is valid and accessible. The nuances of scopes in Python will be discussed in this essay, along with examples that demonstrate their various kinds and behaviours.

- **Global Scope:** The outermost level of a Python program is referred to as the global scope. Functions and other code segments can access variables specified inside this scope.

Ex:- `global_variable = 10`
`def print_global_variable():`
 `print(global_variable)`

`print_global_variable()`
Output: 10

```
1 global_variable = 10
2
3 def print_global_variable():
4     print(global_variable)
5
6 print_global_variable() # Output: 10
7
```



10

- **Local Scope:** A function or a block of code has local scope, and variables specified here are only available inside that particular function or block.

`def example_function():`
 `local_variable = 5`
 `print(local_variable)`

`example_function()`
Output: 5

```
1 def example_function():
2     local_variable = 5
3     print(local_variable)
4
5 example_function() # Output: 5
6
```



5

- **Enclosing Scope(Closure):** An enclosing scope is created when functions are declared inside of other functions. The inner function has access to variables from the outer function.

```
def outer_function():  
    outer_variable = 20  
    def inner_function():  
        print(outer_variable)  
    inner_function()  
outer_function() # Output: 20
```

```
1 def outer_function():  
2     outer_variable = 20  
3  
4     def inner_function():  
5         print(outer_variable)  
6  
7     inner_function()  
8  
9 outer_function() # Output: 20  
10
```

20

- **Built-in Scope:** Without the need for import statements, Python has a number of built-in objects and functions that are accessible throughout the program. Objects like int, str, and methods like print() and len() are all included in this built-in scope.

```
print(len("Hello")) # Output: 5
```

```
1 print(len("Hello"))  
2
```

5

Comprehending scopes in Python is essential for preventing name conflicts, controlling variable visibility, and producing readable and efficient code. With a solid understanding of the scope hierarchy and how they interact, developers may write Python programs that are ordered and easily maintained.

→SECTION 5-- CREATING MULTI-PARAMETER FUNCTION

Programming functions are about more than just enclosing code; they are also about flexibility and handling a wide range of situations. A multi-parameter function is a basic building block that takes several input values and lets programmers write flexible, reusable

code. We will examine the importance of multi-parameter functions and provide appropriate examples to demonstrate their use in this post.

- The range of applications for a single-parameter function might be restricted. Contrarily, multi-parameter functions offer a means of passing and processing many inputs, increasing their versatility and adaptability to a greater variety of use cases.
- The process of defining a multi-parameter function in most programming languages entails passing in many parameters, separated by commas. For values that may be supplied during the function call, these arguments serve as placeholders.

```
def add_numbers(x, y):
```

```
    result = x + y
```

```
    return result
```

The `add_numbers` function in this example accepts two parameters, `x` and `y`, which stand for the numbers that need to be added. The addition's outcome is then returned by the function.

- Because multi-parameter functions may handle a wide range of data types, programmers can design functions that are not limited to a particular kind of input.

```
def concatenate_strings(str1, str2):
```

```
    result = str1 + str2
```

```
    return result
```

- Parameter default values are supported by a wide variety of computer languages, making them optional during function calls. This increases the flexibility of functions with several arguments by enabling some of them to have default values in the event that they are not supplied explicitly.

```
def power(base, exponent=2):
```

```
    result = base ** exponent
```

```
    return result
```

- Multi-parameter functions provide for a wider range of input possibilities, which improves code reusability. Functions that carry out certain tasks can be created by developers and then reused in other sections of their codebase with varying parameter settings.

```
def calculate_volume(length, width, height):
```

```
    volume = length * width * height
```

```
    return volume
```

FACTORIAL OF A NUMBER

```

1- def factorial(n):
2-     if n == 0 or n == 1:
3-         return 1
4-     else:
5-         return n * factorial(n-1)
6-
7- # Get Input from the user
8- num = int(input("Enter a number: "))
9-
10- # Check if the number is non-negative
11- if num < 0:
12-     print("Factorial is not defined for negative numbers.")
13- else:
14-     result = factorial(num)
15-     print(f"The factorial of {num} is {result}")
16-

```

Enter a number: 5
The factorial of 5 is 120
> |

FIBONACCI NUMBERS

```

def generate_fibonacci(limit):
    fibonacci_sequence = [0, 1]

    while fibonacci_sequence[-1] + fibonacci_sequence[-2] <= limit:
        next_fibonacci = fibonacci_sequence[-1] + fibonacci_sequence[-2]
        fibonacci_sequence.append(next_fibonacci)

    return fibonacci_sequence

limit = int(input("Enter the limit for Fibonacci sequence: "))

if limit < 0:
    print("Please enter a non-negative limit.")
else:
    fibonacci_sequence = generate_fibonacci(limit)
    print("Fibonacci sequence up to", limit, "is:", fibonacci_sequence)

```

Enter the limit for Fibonacci sequence: 6
Fibonacci sequence up to 6 is: [0, 1, 1, 2, 3, 5]
>

One essential component of programming that promotes code flexibility, readability, and reusability is the creation of multi-parameter functions. Developers may create more flexible and adaptive software solutions that meet a wider range of needs by creating functions that can handle numerous inputs. This adaptability is essential to the creation of reliable and scalable programming.

→SECTION 6—TUPLES AND DICTIONARIES

Tuples and dictionaries are two strong data structures in Python that provide different functions for handling and arranging data. While dictionaries give a changeable and unordered key-value pairing, tuples offer an immutable and ordered collection of components. The purpose of this essay is to demonstrate the flexibility of tuples and dictionaries in Python programming by going over their features, applications, and use scenarios.

TUPLES:

An ordered set of elements between parenthesis is called a tuple. A tuple is immutable once it is produced; it cannot be changed. Due to their immutability, tuples are frequently used to express fixed sequences of values and are appropriate in situations where data shouldn't be altered.

- **Syntax and Creation:** `my_tuple = (1, 'apple', 3.14, True)`
- **Accessing Elements:** `print(my_tuple[1])` # Output: 'apple'
- **Immutability:** # Attempting to modify a tuple will result in an error

```
my_tuple[0] = 10 # TypeError: 'tuple' object does not support item
assignment
```

- **Use Cases:** Coordinates, RGB colour values, and any other sequence that has to stay consistent throughout the program may all be represented with tuples.

DICTIONARIES:

Python dictionaries are unordered sets of key-value pairs that offer a versatile and effective method of storing and retrieving data. In a dictionary, every key has to be distinct and have a particular value assigned to it.

- **Syntax and Creation:** `my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}`
- **Accessing Elements:** `print(my_dict['age'])` # Output: 25
- **Mutability:** # Modifying a dictionary is allowed
`my_dict['age'] = 26`

- **Use Cases:** # Adding a new key-value pair
`my_dict['occupation'] = 'Engineer'`
Removing a key-value pair
`del my_dict['city']`

To sum up, dictionaries and tuples are essential Python data structures, each with special functions and applications. While dictionaries offer mutability and key-based access, making them ideal for dynamic data mapping, tuples offer immutability and order, making them excellent for fixed sequences. Combining these data structures enables programmers to write expressive and effective solutions for a variety of programming situations.

→SECTION 7—EXCEPTIONS

Within the programming domain, exceptions function as a means of addressing mistakes and unforeseen circumstances that could emerge while code is being executed. Writing robust and fault-tolerant programs requires exception handling, which enables programmers to foresee possible problems and elegantly resolve them. This essay examines the idea of exceptions, explains their use in programming, and offers examples to show how to utilize them.

Events that interfere with a program's usual execution flow are called exceptions. They happen when a software can't continue operating normally due to a mistake or unanticipated situation. The process of recognizing, reacting to, and resolving these unusual occurrences is known as exception handling.

Depending on the type of error, exceptions are categorized into several categories in the majority of programming languages. Typical exclusions consist of:

- **SyntaxError** occurs when a piece of code deviates from the language's syntax specifications.
- When an operation is carried out on an object of the wrong type, a **TypeError** occurs.
- When a function receives an argument of the right type but an incorrect value, a **ValueError** occurs.
- **FileNotFoundError** message appears when a file cannot be opened because it is not in existence.

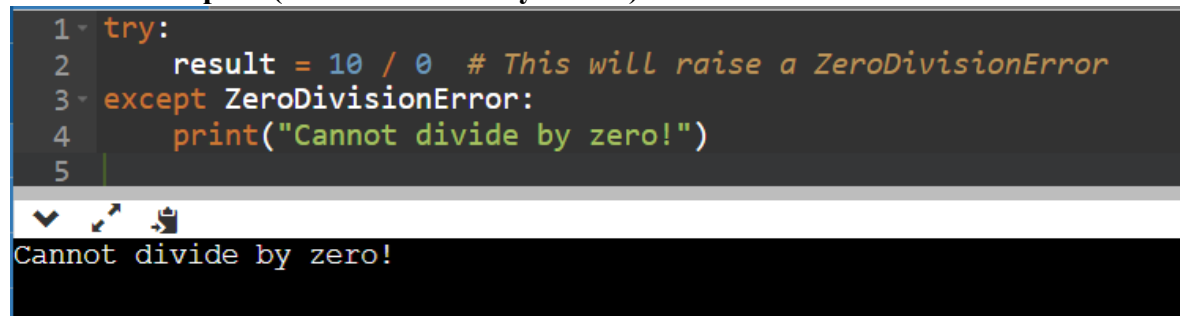
The try-except block is the basic building block for managing exceptions. With this structure, developers may include code that could throw an exception in the "try" block and define the "except" block's handling of the exception.

try:

result = 10 / 0 # This will raise a ZeroDivisionError

except ZeroDivisionError:

print("Cannot divide by zero!")



```

1 try:
2     result = 10 / 0 # This will raise a ZeroDivisionError
3 except ZeroDivisionError:
4     print("Cannot divide by zero!")
5

```

Cannot divide by zero!

If many except clauses are included, a single try-except block can handle multiple exceptions.

try:

user_input = int(input("Enter a number: "))

result = 10 / user_input

except ValueError:

print("Invalid input. Please enter a number.")

except ZeroDivisionError:

print("Cannot divide by zero!")

```

1 try:
2     user_input = int(input("Enter a number: "))
3     result = 10 / user_input
4 except ValueError:
5     print("Invalid input. Please enter a number.")
6 except ZeroDivisionError:
7     print("Cannot divide by zero!")
8

```

```

Enter a number: z
Invalid input. Please enter a number.

```

If the "finally" block is present, it runs whether or not an exception is raised. It is usual practice to utilize this block for cleaning tasks like resource releases or file closures.

```

try:
    file = open("example.txt", "r")
    # Perform operations on the file
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()

```

Developers can use the "raise" keyword to explicitly raise exceptions. This is helpful when there are circumstances that call for an exception.

```

def check_positive(number):
    if number < 0:
        raise ValueError("Number must be positive.")
    return number

```

```

1 def check_positive(number):
2     if number < 0:
3         raise ValueError("Number must be positive.")
4     return number
5 print(check_positive(-1))

```

```

Traceback (most recent call last):
  File "/home/main.py", line 5, in <module>
    print(check_positive(-1))
  File "/home/main.py", line 3, in check_positive
    raise ValueError("Number must be positive.")
ValueError: Number must be positive.

```

Developers can define additional classes that inherit from the basic "Exception" class to generate bespoke exceptions in addition to the built-in exceptions. This makes it possible to handle exceptions in a more detailed manner.

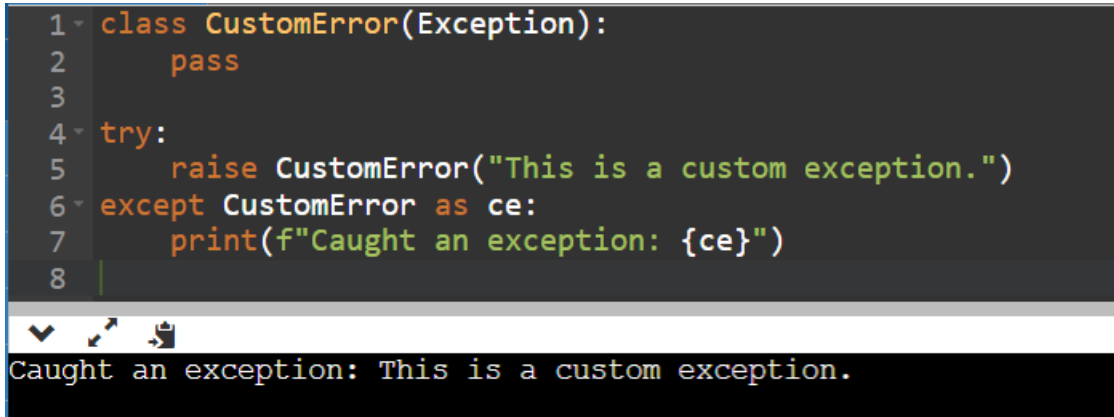
```

class CustomError(Exception):
    pass

```



```
try:
    raise CustomError("This is a custom exception.")
except CustomError as ce:
    print(f"Caught an exception: {ce}")
```



```
1 class CustomError(Exception):
2     pass
3
4 try:
5     raise CustomError("This is a custom exception.")
6 except CustomError as ce:
7     print(f"Caught an exception: {ce}")
8
Caught an exception: This is a custom exception.
```

Programming's essential tool, exceptions give programmers a way to elegantly manage mistakes and unforeseen circumstances. Try-except blocks help programmers build more dependable and robust applications. Exception handling helps with debugging and preserving code integrity in addition to making the user experience better by preventing crashes. Writing reliable software solutions requires you to become an expert in exception management.

→MODULE TEST-4—COMPLETION TEST

It mainly involved a series of solving questions rather than subjective questions, and most part was from the “Function” part.

As we have progressed from functions to exceptions, we have looked at basic ideas that are the foundation of programming in many languages. As modular chunks of code, functions offer an organized method for grouping and reusing logic. We may empower ourselves to write effective, legible, and modular programs by being knowledgeable about functions.

As we move on to the topic of functions returning values, we see how important it is for functions to inform the caller code of their outcomes. Return values make it possible to package calculations, which encourages code reuse and improves our applications' modularity.

Python scopes have become essential for comprehending how variables are accessed and modified across various code segments. Understanding the notions of closures, global scope, and local scope can help developers control variable visibility and avoid unwanted side effects.

Tuples and dictionaries were examined to show off how flexible these Python data structures are. Due to their ordered and immutable structure, tuples provide stability for data sequences that are fixed. Conversely, dictionaries offer a flexible and effective way to retrieve and organize data because of their modifiable key-value pairings.

Ultimately, we explored exceptions, a critical component of creating reliable and resilient programs. We can anticipate mistakes and unforeseen circumstances and respond to them with grace when we comprehend and manage exceptions. The try-except block gives developers strong capabilities to handle unusual occurrences and build robust software, coupled with the finally block and custom exceptions.

In conclusion, the path of programming is characterized by an ongoing pursuit of robustness, clarity, and modularity, from the fundamental ideas of functions to the intricate details of exceptions. Acquiring knowledge of these ideas improves code quality and gives engineers the ability to confidently and precisely negotiate the challenges of software development.

CONCLUSION

From the very beginnings of programming principles to the complex field of exception management, our path has been a rich tapestry of learning and exploration. We started with comprehending functions, which are the basic building elements. Together with giving us the ability to encapsulate functionality, these modular entities also opened us new possibilities for code structure and reusability. Our code got more versatile by making function returns the currency of dynamic interactions.

We learned the skill of changeable visibility as we explored the complex realm of scopes. Python's global and local scopes became the fundamental building blocks of the architecture, averting accidental clashes and strengthening our comprehension of the access and manipulation of data. This understanding paved the way for the investigation of flexible data structures. For fixed sequences, tuples' immutable and ordered structure offered stability, but dictionaries' changeable key-value pairs gave our data organizing efforts some dynamism.

The art of exception handling resonated with the crescendo of our investigation. This crucial area of programming sharpened our abilities to foresee mistakes and react tactfully to unforeseen circumstances. As a dependable protector against the unpredictable nature of code execution, the try-except block has emerged, enabling a more resilient and fault-tolerant software environment.

Through this extensive journey, we have not only gained information, but also developed a deep comprehension of the fundamental ideas that support dependable and effective software development. Equipped with the capacity to write well-structured, logical, and robust code, our experience serves as a monument to the complex terrain of programming, which we can now traverse with assurance and skill.

CERTIFICATE



Statement of Achievement

K V K Sai Bhaskar

has successfully achieved student level credential for completing the Python Essentials 1 course, provided by Cisco Networking Academy in collaboration with OpenEDG Python Institute.

The graduate is able to proficiently:

- Design, develop, debug, execute, and refactor simple computer programs written in Python 3.
- Think algorithmically to analyze problems and implement them as computer processes.
- Use the syntax, semantics, and the most important elements of the Python Standard Library to write Python scripts and resolve typical implementation challenges.
- Understand the role of a programmer in the software development process.
- Attempt the qualification PCEP - Certified Entry-Level Python Programmer from OpenEDG Python Institute and continue their professional development at an intermediate level with Python Essentials 2.



Scan to Verify

Laura Quintana

Laura Quintana
Vice President and General Manager
Cisco Networking Academy

May 20, 2023