Shape your dreams
with
C-DAC's
Multilingual Training Programs

Python

Affordable multilingual Computer Education with integrated Graphics and Intelligence based Script Technology (GIST) from C-DAC

Index

Chapter 1

Python Introduction

1.1 What is Python ?

Python is a versatile and powerful programming language that has become increasingly popular in recent years. It was created by Guido van Rossum in the late 1980s and first released in 1991. Python's popularity can be attributed to its simplicity, readability, and ease of use.

One of the main advantages of Python is its flexibility. It can be used for a variety of purposes, including web development, software development, mathematics, system scripting, and data science. Python is also platform-independent, meaning it can run on various operating systems such as Windows, Linux, and macOS.

Python is an interpreted language, which means that you can run your program without the need to compile it first. This makes it easy for developers to write and test code quickly. Python is also interactive, allowing you to write and execute code directly from the command line or through an interactive interpreter.

Python is an object-oriented language, which means that it supports the encapsulation of code within objects. This allows developers to write code that is more modular and easier to maintain. Python also supports other programming paradigms, such as functional and procedural programming.

Another advantage of Python is its beginner-friendly syntax, which makes it easy for new programmers to learn. Python's simple and elegant syntax is often described as being similar to pseudocode, which makes it easy to understand and write.

Python has a large standard library, which provides pre-built modules and functions that can be used to simplify common programming tasks. In addition, Python has a rich ecosystem of third-party libraries and frameworks, such as Django, Flask, NumPy, Pandas, and TensorFlow, which help developers build complex applications quickly and efficiently.

Python is widely used by tech giants such as Google, Instagram, and YouTube, making it a valuable skill for aspiring developers. It has become one of the most popular programming languages for data science and machine learning due to its powerful libraries and frameworks.

In conclusion, Python is a versatile, powerful, and easy-to-learn programming language that can be used for a variety of applications. Its simplicity, readability, and strong community support make it an excellent choice for both beginner and experienced developers.

1.2 What can Python do ?

Python is a versatile and powerful programming language that can be used for a wide variety of tasks. It has gained popularity due to its simplicity, ease of use, and flexibility.

One of the primary uses of Python is for server-side web application development. Python has a number of frameworks, such as Django and Flask, that make it easy to create web applications quickly and efficiently. Python's syntax is easy to read and understand, which makes it a popular choice for web developers.

Python can also be used to create workflows alongside other software. It has strong integration capabilities with other programming languages, such as Java and C++, which makes it a popular choice for developers who need to work across multiple languages.

Python can connect to a wide variety of database systems, including MySQL, PostgreSQL, and MongoDB. It can also read and modify files, making it a useful tool for data manipulation and analysis.

Python is particularly useful for handling big data and performing complex mathematical operations. It has powerful libraries, such as NumPy and Pandas, that make it easy to work with large datasets and perform statistical analysis.

Python supports a variety of programming paradigms, including functional and structured programming as well as object-oriented programming (OOP). This makes it a versatile language that can be used for a wide range of applications, from rapid prototyping to production-ready software development.

Python is also highly flexible in terms of how it can be used. It can be used as a scripting language for automating tasks or as a compiled language for building large applications. It provides high-level dynamic data types and supports dynamic type checking, which makes it easy to work with.

Python also supports automatic garbage collection, which means that developers don't need to worry about manually managing memory. Finally, Python can be easily integrated with other programming languages, including C, C++, COM, ActiveX, CORBA, and Java.

1.3 Why Python ?

Python is a popular choice among developers for many reasons. One of the main advantages of Python is its ability to work on different platforms, including Windows, Mac, Linux, and Raspberry Pi. This makes it a versatile language that can be used for a wide range of applications.

Another key advantage of Python is its simple and easy-to-read syntax. Python code resembles the English language, which makes it easy to understand and write code quickly. In addition, Python's syntax allows developers to write programs with fewer lines of code than some other programming languages. This can help to increase productivity and make it easier to maintain and update code over time.

Python runs on an interpreter system, which means that code can be executed as soon as it is written. This allows developers to quickly prototype and test ideas, which can

be particularly useful in the early stages of a project. Python's interpreter system also makes it easy to work with and learn for beginners.

Python is a versatile language that can be used in a variety of programming paradigms, including procedural, object-oriented, and functional programming. This flexibility allows developers to choose the approach that best suits their needs and the requirements of their projects.

Overall, Python's versatility, simplicity, and flexibility make it a popular choice among developers for a wide range of applications, from web development to scientific computing to machine learning.

1.4 Good to know

Python is an actively developed and widely used programming language. Its most recent major version is Python 3, which is the version we will be using in this tutorial. Although Python 2 is still popular, it is no longer being updated with new features and is only receiving security updates.

When working with Python, you have the option of writing code in a text editor or an Integrated Development Environment (IDE). IDEs like Thonny, PyCharm, Netbeans, and Eclipse are particularly useful for managing large collections of Python files.

One of Python's strengths is its readability. Python code is designed to be easy to read and understand, with syntax that resembles the English language and is influenced by mathematics. For example, Python uses new lines to complete a command, rather than relying on semicolons or parentheses like some other programming languages.

Python also uses indentation to define the scope of loops, functions, and classes, rather than using curly-brackets like many other programming languages. This approach to indentation-based scoping can make Python code easier to read and less error-prone, but can take some getting used to if you are coming from another programming language.

1.5 History of Python

Guido van Rossum began developing Python in December 1989 as a hobby project while working at the National Research Institute for Mathematics and Computer Science in the Netherlands. He was inspired by ABC, a general-purpose programming language, and set out to create a more intuitive and easy-to-learn language with similar features.

The first version of Python, version 0.9.0, was released in February 1991. Over the next few years, Guido continued to work on Python and released several major versions, with Python 2.0 being released in 2000.

In 2008, development of Python 3.0 began, with the goal of addressing some of the design flaws in Python 2.x and improving the language overall. Python 3.0 was released in 2008, and since then, all new development efforts have been focused on Python 3.x.

Python has continued to grow in popularity and is now one of the most widely used programming languages in the world. It is used by individuals and organizations for a wide range of purposes, including web development, data analysis, scientific computing, and artificial intelligence.

1.6 Python Features

Python has several features that make it a popular programming language:

Easy-to-learn: Python has a simple syntax, few keywords, and a clear structure. This makes it easy for beginners to pick up the language quickly.

Easy-to-read: Python code is designed to be easily read and understood. It is more visually defined, making it easier to comprehend at a glance.

Easy-to-maintain: Python's source code is organized and easy-to-maintain, which is helpful for programmers who need to modify and update code over time.

Broad standard library: Python's standard library is extensive and cross-platform compatible, making it easy to work with a variety of different systems and platforms.

Interactive Mode: Python's interactive mode allows for easy testing and debugging of code snippets, which can help with development and troubleshooting.

Portable: Python can run on a wide variety of hardware platforms, with the same interface on all platforms.

Extendable: Python allows for the addition of low-level modules to the interpreter, enabling programmers to customize their tools and make them more efficient.

Databases: Python provides interfaces to all major commercial databases, making it a useful tool for working with data.

GUI Programming: Python supports the creation of GUI applications that can be easily ported to different systems, including Windows, Macintosh, and Unix-based systems.

Scalable: Python provides a better structure and support for large programs than shell scripting, making it a powerful tool for developing complex applications.

1.7 Python 2 vs. Python 3

In most programming languages, whenever a new version releases, it supports the features and syntax of the existing version of the language, therefore, it is easier for the projects to switch to the newer version. However, in the case of Python, the two versions Python 2 and Python 3 are very much different from each other.

A list of differences between Python 2 and Python 3 are given below:

No.	Python 2	Python 3
1	Python 2 uses print as a statement and uses print "something" to print some string on the console.	Python 3 uses print as a function and uses print("something") to print something on the console.
2	Python 2 uses the function raw_input() to accept the user's input. It returns the string representing the value, which is typed by the user. To convert it into the integer, we need to use the int() function in Python.	Python 3 uses the input() function which automatically interprets the type of input entered by the user. However, we can cast this value to any type by using primitive functions (int(), str(), etc.).
3	Python 2, the implicit string type is ASCII.	Python 3, the implicit string type is Unicode.
4	Python 2 contains the xrange() function. The xrange() is the variant of the range() function which returns a xrange object that works similar to the Java iterator.	In Python 3 the xrange() function of Python 2 is replace by the range() function. The range() returns a list, for example the function range(0,3) contains 0, 1, 2.
5	In Python 2, you can raise an exception using the raise statement with an optional comma-separated tuple of values, like this: **'raise ExceptionType, value'**.	In Python 3, you can only raise an exception using the raise statement with a single argument, like this: **'raise ExceptionType(value)'**.

1.8 Compiler and Interpreter

A compiler and an interpreter are both computer programs that are used to translate high-level programming languages into low-level machine code that a computer can understand and execute. However, they differ in the way they carry out this translation process.

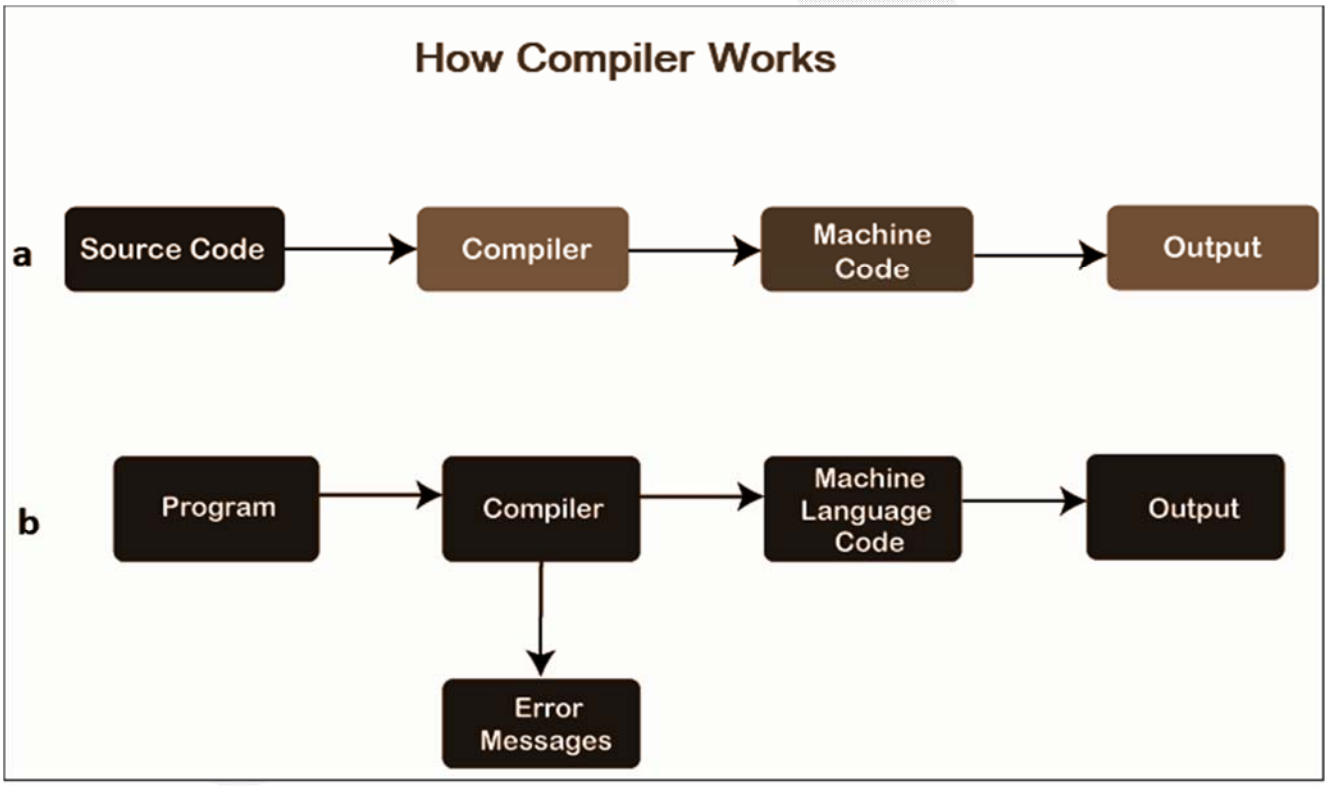
If the goal is to have faster and more efficient code that will be executed multiple times, a compiler is usually the better choice. If the goal is more flexible and dynamic execution, or if the code will only be executed a few times, an interpreter is usually the better choice.

Compiler :

A compiler is a program that reads an entire program in a high-level language and translates it into an equivalent program in a low-level language, such as machine language or assembly language. The compiled program is then executed directly by the computer's CPU. Compiling a program typically involves several stages, including lexical analysis, syntax analysis, code generation, and optimization. The resulting low-level code is usually stored in a separate file, which can be executed at a later time.

Advantages of using a compiler include faster execution times, as the entire program is translated in one go, and the ability to optimize the code for better performance. Additionally, a compiled program can be distributed and executed on any computer that has the appropriate runtime environment.

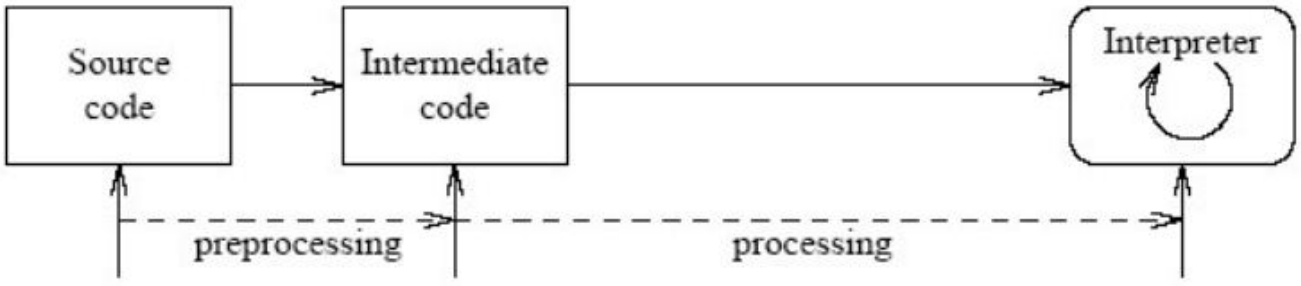
A compiler is often used when the program needs to be executed repeatedly, or when a program needs to be run on a different computer or platform than the one it was developed on. Compilers take more time to initially translate the code, but once the code has been compiled, it can be executed much faster than interpreted code. Compiled code is also generally more efficient in terms of memory usage and CPU utilization.



Interpreter :

An interpreter, on the other hand, is a program that reads and executes code one line at a time. Instead of generating low-level machine code, an interpreter reads each line of code and performs the necessary actions on the computer. Interpreted programs do not need to be compiled before they can be executed, and can be modified and executed immediately. However, interpreting code is generally slower than executing compiled code, as each line of code needs to be translated and executed separately.

Advantages of using an interpreter include the ability to quickly test and modify code, easier debugging, and platform independence, as interpreted code can be executed on any machine with the appropriate interpreter. Additionally, interpreted languages often have a simpler syntax, making them easier to learn for beginners.



Interpretation

An interpreter is often used for tasks such as scripting or prototyping, where the code is executed only once or a few times. Interpreters take less time to initially translate the code, and are more flexible in terms of debugging and error checking. They are also often used for languages that require more dynamic execution, such as in web development where code needs to be interpreted on the fly.

1.9 Which environment for you ?

Different environments have different use cases and are preferred by different groups of users:

- **Anaconda**: It is commonly used by data scientists and machine learning engineers for scientific computing, data analysis, and machine learning tasks. It comes with pre-installed packages and libraries required for these tasks, and users can easily create and manage different environments with different configurations.
- **pip**: It is the default package manager for Python and is used by developers to install and manage packages for their projects. It is widely used in web development, software development, and automation tasks.
- **Virtualenv**: It is used by developers to create isolated Python environments for their projects, which allows them to install and manage packages without affecting the global Python environment. It is commonly used in web development and software development.
- **PyCharm**: It is an integrated development environment (IDE) that is commonly used by Python developers for software development, web development, and data science tasks. It provides advanced features such as code completion, debugging, testing, and version control integration.
- **Jupyter Notebook:** It is commonly used by data scientists and researchers for data exploration, visualization, and analysis. It allows users to create interactive notebooks that combine code, text, and visualizations, which can be shared and executed by others.

- **Spyder:** It is an IDE that is commonly used by scientific Python users for data analysis and scientific computing tasks. It provides features such as variable explorer, data viewer, and integrated plotting, which are useful for these tasks.

1.10 Python Interpreter

The command-line interpreter for Python is called the Python interpreter. It allows you to execute Python code and interact with the interpreter directly through the command-line interface.

To use the Python interpreter, open a command prompt or terminal window and type python followed by the Enter key. This will launch the Python interpreter and display the version number and some copyright information.

Once you are in the Python interpreter, you can enter Python code directly and the interpreter will execute it immediately. For example, you can type print("Hello, world!") and the interpreter will print the message "Hello, world!" to the screen.

You can also use the interpreter to run Python scripts saved as files on your computer. To do this, you can type python <filename.py> where <filename.py> is the name of the Python script file. The interpreter will execute the code in the file and display any output or errors.

In addition to the standard Python interpreter, there are also alternative Python interpreters available that offer additional features or functionality. Examples include IPython, Jupyter, and bpython.

Chapter 2

Fundamentals of Python

2.1 Print "Hello, world!"

1. Open the command prompt or terminal on your computer.
2. Type `python` and hit Enter to launch the Python interpreter.
3. Type `print("Hello, world!")` and hit Enter.
4. The output should be Hello, World!.

You should see the output `Hello, world!` printed on the next line.

```
>>> print("Hello, World!")
 Hello, World!
>>>
```

2.2 About Variables

It's important to note that when we define variables in Python, we don't need to specify the data type explicitly. Python is a dynamically-typed language, which means that the data type of a variable is determined automatically based on the value assigned to it. In this case, `a` and `b` are both integers, so Python automatically assigns the `int` data type to these variables.

```
>>> a = 10
>>> b = 20
>>> print(a)
10
>>> print(b)
20
```

In the above example, we first defined two integer variables `**a**` and `**b**` and assigned the values of **5** and **10** respectively. Then we used the `**print()**` function to display the values of `a` and `b` on the console. When we ran this code in the Python interpreter, it printed the output as `5` and `10` on separate lines.

2.3 Example of addition and subtraction program in the Python interpreter

In this example, we created two integer variables 'a' and 'b' and performed addition and subtraction operations using the '+' and '-' operators, respectively. We then stored the results in variables 'c' and 'd' and printed them using the print() function. Finally, we exited the interpreter.

1. Open the Python interpreter by typing "python" in the command prompt or terminal.
2. Create two integer variables, for example:

```
>>> a = 10
>>> b = 5
```

3. To perform addition, use the '+' operator:

```
>>> c = a + b
>>> print(c)
```

4. To perform subtraction, use the '-' operator:

```
>>> d = a - b
>>> print(d)
```

5. Exit the interpreter by typing **"exit()"** or pressing **Ctrl+z**.

2.4 Python Interpreter limitation

Command line interpreter, also known as a command prompt, has some limitations when it comes to using it for programming. Some of the limitations are:

1. Lack of Graphical User Interface (GUI): Command line interpreter lacks a graphical user interface, which makes it difficult to interact with the program. Users need to rely on their knowledge of commands and syntax to perform actions.

2. Limited Editing Capabilities: Command line interpreter has limited editing capabilities compared to Integrated Development Environments (IDEs) such as PyCharm or Visual Studio Code. Users can only make simple edits to their code, and there is no auto-completion or code highlighting.

3. No debugging tools: Command line interpreter does not have debugging tools built-in. This means that users have to rely on print statements or other debugging techniques to debug their code.

4. Lack of Code Reusability: Command line interpreter does not support code reusability as it does not allow users to create reusable modules or libraries.

5. Steep Learning Curve: Command line interpreter has a steep learning curve, especially for beginners who are not familiar with the command line interface. It requires users to memorize commands and syntax, which can be overwhelming at first.

Overall, while the command line interpreter is a useful tool for quick and simple programming tasks, it has some limitations that make it unsuitable for more complex projects.

I'm assuming you are asking about why one might avoid using the command-line interpreter for certain tasks.

While the command-line interpreter can be useful for quick, one-off tasks or testing small code snippets, it may not be the best tool for all programming tasks.

Here are some reasons why one might avoid using the command-line interpreter:

Limited functionality: The command-line interpreter has limited functionality compared to integrated development environments (IDEs) or code editors. It may not have features such as code completion, syntax highlighting, and debugging tools that can make programming more efficient.

Difficult to manage large projects: The command-line interpreter is not well-suited for managing large code projects. It can be difficult to keep track of multiple files and dependencies using the command-line interface.

Limited collaboration features: The command-line interpreter does not provide easy collaboration features like version control or integrated communication tools.

Steep learning curve: The command-line interpreter can have a steep learning curve for those new to programming or those who are not familiar with the command-line interface.

Limited graphical user interface (GUI): The command-line interpreter does not have a graphical user interface (GUI) like an IDE or code editor. This can make it difficult to visualize and interact with the code in a meaningful way.

2.5 Operators in python

An operator is a symbol or a keyword that performs specific operations on one or more operands and produces a result. In programming, operators are used to manipulate data values and variables to perform different computations or comparisons. For example, arithmetic operators such as +, -, *, / perform mathematical calculations, and comparison operators such as >, <, ==, != are used to compare values and return Boolean results (True or False). Python supports various types of operators such as arithmetic, comparison, assignment, logical, bitwise, and membership operators.

Arithmetic Operators		
Operator	**Operation**	**Example**
+	Addition	5 + 3 = 8
-	Subtraction	5 - 3 = 2
*	Multiplication	5 * 3 = 15
/	Division	5 / 3 = 1.66667
%	Modulus (remainder of division)	5 % 3 = 2
**	Exponentiation	5 ** 3 = 125
//	Floor division (division rounding down to nearest whole number)	5 // 3 = 1

Comparison Operators		
Operator	**Description**	**Example**
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true
!=	If values of two operands are not equal, then the condition becomes true.	(a != b) is true
>	If the value of the left operand is greater than the value of the right operand, then the condition becomes true.	(a > b) is not true
<	If the value of the left operand is less than the value of the right operand, then the condition becomes true.	(a < b) is true
>=	If the value of the left operand is greater than or equal to the value of the right operand, then the condition becomes true.	(a >= b) is not true
<=	If the value of the left operand is less than or equal to the value of the right operand, then the condition becomes true.	(a <= b) is true

Logical Operators			
Operator	**Description**	**Example**	**Result**
and	Returns True if both statements are true	True and False	False
or	Returns True if at least one of the statements is true	True or False	True
not	Inverts the result, returns False if the result is true	not True	False

For example:

```
a = 5
b = 10
c = 15

print(a < b and b < c)  # Output: True
print(a < b or b > c)  # Output: True
print(not(a < b and b < c))  # Output: False
```

Identity Operator		
Operator	**Description**	**Example**
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Membership Operators		
Operator	**Description**	**Example**
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Bitwise Operators		
Operator	**Description**	**Example**
&	Bitwise AND: Returns 1 if both bits are 1	5 & 3 = 1
|	Bitwise OR: Returns 1 if either bit is 1	5 | 3 = 7
^	Bitwise XOR: Returns 1 if only one of the bits is 1	5 ^ 3 = 6
~	Bitwise NOT: Inverts all the bits	~5 = -6
<<	Bitwise Left Shift: Shifts the bits to the left by n	5 << 2 = 20
>>	Bitwise Right Shift: Shifts the bits to the right by n	5 >> 2 = 1

Assignment Operators		
Operator	**Example**	**Equivalent to**
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
//=	x //= 5	x = x // 5
%=	x %= 5	x = x % 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
<<=	x <<= 2	x = x << 2
>>=	x >>= 2	x = x >> 2

2.6 Data Types

Data types		
Data Type	**Description**	**Example**
Text Type (str)	Used for representing text in Python. Enclosed in either single or double quotes.	name = "John"
Numeric Types	Used for representing numbers in Python. Integers (int) are used for whole numbers, floating-point numbers (float) are used for decimals, and complex numbers (complex) are used for numbers with both a real and imaginary part.	age = 25 price = 9.99 complex_num=3+ 4j
Sequence Types	Used for representing sequences of elements. Lists (list) are mutable sequences that can contain elements of any data type. Tuples (tuple) are immutable sequences that can also contain elements of any data type. Ranges (range) are immutable sequences of numbers.	my_list = [1, 2, 3] my_tuple = ("apple", "banana", "cherry") my_range = range(5)
Mapping Type	Used for representing mappings of keys to values. Dictionaries (dict) are mutable mappings where each element is a key-value pair.	my_dict = { "name":"John", "age": 25}
Set Types	Used for representing sets of unique elements. Sets (set) are mutable sets that contain unique elements of any data type. Frozensets (frozenset) are immutable sets that also contain unique elements of any data type.	my_set = {1, 2, 3} my_frozenset = frozenset({4, 5, 6})
Boolean Type	Used for representing True or False values. Booleans (bool) are used for logical operations and comparisons.	is_sunny = True
Binary Types	Used for representing sequences of bytes. Bytes (bytes) are immutable sequences of bytes. Bytearrays (bytearray) are mutable sequences of bytes. Memoryviews (memoryview) are used to access the memory of other objects in a flexible way.	my_bytes = b"Hello" my_bytearray = bytearray(5) my_memoryview = memoryview(b"Hello")
None Type	Used for representing the absence of a value. None is a built-in constant that represents nothing.	my_var = None

2.7 User input

In Python, the input() function allows the user to enter values from the keyboard, which can then be used in the program for further processing. When input() is called, the program stops and waits for the user to enter text, which is then returned as a string. This string can be stored in a variable and used in various ways, such as for mathematical calculations or for displaying messages to the user. The input() function can also be used in conjunction with conditional statements and loops to create more complex programs that take user input and respond accordingly. However, it's important to be cautious when using input(), as it can also be used for malicious purposes such as injecting code or stealing sensitive information.

Example

```
user_input = input("Enter some text: ")
print("You entered:", user_input)
```

When this script is run, the input function prompts the user to enter some text, which is then stored in the user_input variable.

2.8 String formatting

String formatting in Python is the process of constructing strings that can contain placeholders or variables that are dynamically replaced with their corresponding values at runtime. It involves various techniques such as using format function, f-string, raw string, different types of strings, concatenation, and exception characters to create well-formatted and easily readable strings. Proper use of string formatting techniques can help make code more efficient and maintainable.

In Python, there are three types of **string literals:**

- single-quoted strings ('...'),
- double-quoted strings ("..."),
- triple-quoted strings ('''...''' or """...""").

Single and double quotes are functionally equivalent, but triple quotes are used to represent strings that span multiple lines.

String Concatenation :

String concatenation is the process of joining two or more strings together. In Python, it can be done using the **'+'** operator.

Example:

```
str1 = "Hello"
str2 = "World"
str3 = str1 + " " + str2
print(str3)

# Output: "Hello World"
```

Escape Characters :

Escape characters are used to represent certain characters that cannot be typed as-is in a string. For example, '\n' is used to represent a newline character. Other commonly used escape characters include '\t' for a tab character and '\' for a backslash character.

Example:

```
print("Hello\nWorld")
# Output:
# Hello
#   World
```

Example:

```
print("C:\\Users\\")
# Output: C:\Users\
```

Character	Description
\n	Newline
\t	Tab
\\	Backslash
\'	Single quote
\"	Double quote

Format Function :

The **format**() function is used to insert values into a string. It takes one or more arguments, which are inserted into the string at the specified locations using curly braces {} as placeholders.

Example :

```
name = "John"
age = 30
print("My name is {} and I am {} years old".format(name, age))

# Output: My name is John and I am 30 years old
```

F-Strings :

F-strings are a new way of formatting strings in Python 3.6 and later. They allow you to embed expressions inside string literals, using curly braces {}.

Example :

```
name = "John"
age = 30
print(f"My name is {name} and I am {age} years old")

# Output: My name is John and I am 30 years old
```

Raw String :

A raw string is a string that is prefixed with the letter 'r' or 'R'. It tells Python to interpret the string literally, without any special characters.

Example :

```
print(r"C:\Users\")

 # Output: C:\Users\
```

2.9 Type Casting

Type casting refers to the process of changing the data type of a variable from one type to another. In Python, type casting can be done using built-in functions such as int(), float(), str(), bool(), etc. Type casting is useful when we want to perform operations or comparisons on variables of different data types, or when we want to convert user input (which is usually in the form of strings) into numeric values for calculations. It is important to note that not all data types can be cast into other data types, and some conversions may result in data loss or errors.

Example :

```
# integer to string
num_int = 123
num_str = str(num_int)
print("num_int as string:", num_str)

# string to integer
num_str = "456"
num_int = int(num_str)
print("num_str as integer:", num_int)

# float to integer
num_float = 3.14
num_int = int(num_float)
print("num_float as integer:", num_int)

# integer to float
num_int = 100
num_float = float(num_int)
print("num_int as float:", num_float)
```

Chapter 3

List, Tuple, Dictionary, Set, Forzenset

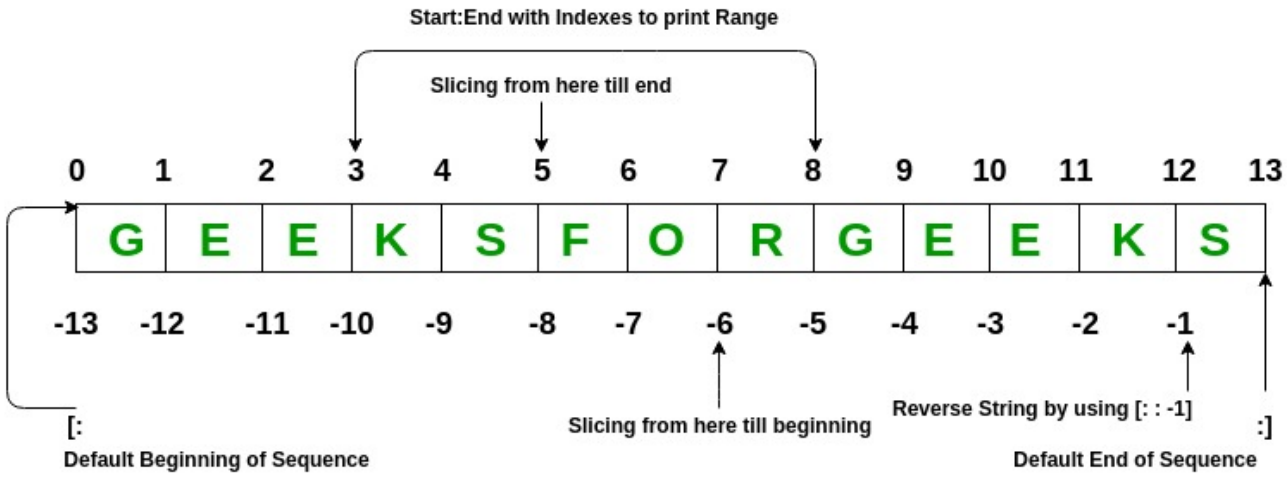
3.1 List

A collection of ordered and mutable elements, represented by square brackets []. Some of its methods are append(), insert(), remove(), pop(), count(), index(), sort(), and reverse().

Example :

```
my_list = [1, 2, 3, 4, 5]
my_list.append(6)
print(my_list)

# Output: [1, 2, 3, 4, 5, 6]
```



In Python, lists are a collection of items that can be indexed and sliced to retrieve a subset of the list. Indexing is the process of accessing a specific element within a list using its position within the list. Slicing is a method of retrieving a subset of elements from a list using the colon operator (:).

For **example**, let's say we have a list of numbers:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We can use indexing to access specific elements within the list, like so:

```
print(numbers[0])   # Output: 1
print(numbers[5])   # Output: 6
print(numbers[-1])  # Output: 10 (accessing last element using negative index)
```

We can also use slicing to retrieve a subset of elements from the list, like so:

```
print(numbers[2:6])   # Output: [3, 4, 5, 6]
print(numbers[:5])    # Output: [1, 2, 3, 4, 5]
print(numbers[5:])    # Output: [6, 7, 8, 9, 10]
print(numbers[2:9:2]) # Output: [3, 5, 7, 9] (using step value to retrieve every other element)
```

Nested lists are lists that contain other lists as elements. We can use indexing and slicing to access elements within nested lists as well.

For example, let's say we have a nested list of student grades:

```
grades = [['John', 85], ['Emma', 92], ['Alex', 77], ['Mia', 88]]
```

We can use indexing and slicing to access specific elements within the nested list, like so:

```
print(grades[0])      # Output: ['John', 85]
print(grades[1][1])  # Output: 92 (accessing grade for Emma)
print(grades[:2])   # Output: [['John', 85], ['Emma', 92]] (slicing to retrieve first two elements)
```

List methods are built-in functions in Python that can be used to manipulate and work with list objects. They allow for easy insertion, deletion, retrieval, and modification of list elements. There are many list methods available in Python, each designed to perform specific operations on lists.

1. `append`: Adds an element to the end of the list.

Example:

```
fruits = ['apple', 'banana', 'cherry']
fruits.append('orange')
print(fruits)

# Output: ['apple', 'banana', 'cherry', 'orange']
```

2. `**extend**`: Adds all elements of a list to the end of another list.

Example:

```
fruits = ['apple', 'banana', 'cherry']
more_fruits = ['orange', 'kiwi', 'mango']
fruits.extend(more_fruits)
print(fruits)

# Output: ['apple', 'banana', 'cherry', 'orange', 'kiwi', 'mango']
```

3. `**insert**`: Adds an element at a specific index in the list.

Example:

```
fruits = ['apple', 'banana', 'cherry']
fruits.insert(1, 'orange')
print(fruits)

# Output: ['apple', 'orange', 'banana', 'cherry']
```

4. `**remove**`: Removes the first occurrence of a specified element from the list.

Example:

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove('banana')
print(fruits)

# Output: ['apple', 'cherry']
```

5. `**pop**`: Removes the element at the specified index and returns it.

Example

```
fruits = ['apple', 'banana', 'cherry']
removed_fruit = fruits.pop(1)
print(removed_fruit)  # Output: 'banana'
print(fruits)  # Output: ['apple', 'cherry']
```

6. `**index**`: Returns the index of the first occurrence of a specified element in the list.

```
fruits = ['apple', 'banana', 'cherry']
index = fruits.index('banana')
print(index)

 # Output: 1
```

7. `**count**`: Returns the number of occurrences of a specified element in the list.

```
fruits = ['apple', 'banana', 'cherry', 'banana']
count = fruits.count('banana')
print(count)
# Output: 2
```

8. `**sort**`: Sorts the elements of the list in ascending order.

```
numbers = [4, 2, 6, 3, 1, 5]
numbers.sort()
print(numbers)
# Output: [1, 2, 3, 4, 5, 6]
```

9. ‘**reverse**`: Reverses the order of the elements in the list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.reverse()
print(fruits)
 # Output: ['cherry', 'banana', 'apple']
```

10. `**copy()**`: Returns a copy of the list

```
my_list = [1, 2, 3, 4, 5]
new_list = my_list.copy()
print(new_list)
# Output: [1, 2, 3, 4, 5]
```

11. `clear()`: Removes all elements from the list

Example:

```
my_list = [1, 2, 3, 4, 5]
my_list.clear()
print(my_list)  # Output: []
```

3.2 Tuple

Tuples are similar to lists, but they are **immutable**, meaning their elements cannot be modified once they are created. They are typically used to group related data together. Tuples are created using parentheses and can contain elements of any data type.

1. Creating a tuple: A tuple can be created using round brackets `()` with or without elements separated by commas. For example:

```
# Creating an empty tuple
empty_tuple = ()
```

```
# Creating a tuple with elements
fruit_tuple = ('apple', 'banana', 'orange')
```

2. Accessing elements of a tuple: Elements of a tuple can be accessed using indexing and slicing. Indexing starts from 0 and goes up to the length of the tuple minus one. Slicing can be done using the colon `:` operator. For example:

```
# Accessing the first element of the tuple
fruit_tuple = ('apple', 'banana', 'orange')
```

```
print(fruit_tuple[0])  # Output: 'apple'
```

```
# Accessing a slice of the tuple
print(fruit_tuple[1:3])  # Output: ('banana', 'orange')
```

3. Changing elements of a tuple: A tuple is immutable, which means its elements cannot be changed once it is created. If we try to change an element of a tuple, we will get a `TypeError`. For example:

```
fruit_tuple = ('apple', 'banana', 'orange')
fruit_tuple[0] = 'pear'  # TypeError: 'tuple' object does not support item assignment
```

4. Converting a tuple to a list: A tuple can be converted to a list using the `list()` function. For example:

```
fruit_tuple = ('apple', 'banana', 'orange')
fruit_list = list(fruit_tuple)
print(fruit_list)  # Output: ['apple', 'banana', 'orange']
```

5. Counting the occurrence of an element in a tuple: The `count()` method can be used to count the number of times an element occurs in a tuple. For example:

```
fruit_tuple = ('apple', 'banana', 'orange', 'apple', 'banana')
print(fruit_tuple.count('apple'))  # Output: 2
```

6. Finding the index of an element in a tuple: The `index()` method can be used to find the index of the first occurrence of an element in a tuple. For example:

```
fruit_tuple = ('apple', 'banana', 'orange')
print(fruit_tuple.index('banana'))  # Output: 1
```

7. Using nested tuples: Tuples can contain other tuples as elements, creating nested tuples. For example:

```
fruit_tuple = ('apple', ('banana', 'orange'))
print(fruit_tuple[1][0])  # Output: 'banana'
```

8. **len()**: returns the number of items in the tuple.

Example

```
fruits = ('apple', 'banana', 'orange', 'apple')
length = len(fruits)
print(length)  # Output: 4
```

9. sorted(): returns a sorted list of the specified tuple. Example

```
fruits = ('apple', 'banana', 'orange', 'cherry')
sorted_fruits = sorted(fruits)
print(sorted_fruits)  # Output: ['apple', 'banana', 'cherry', 'orange']
```

10. max(): returns the item with the highest value in the tuple. Example

```
numbers = (1, 5, 3, 8, 2)
max_num = max(numbers)
print(max_num)  # Output: 8
```

11. min(): returns the item with the lowest value in the tuple. Example

```
numbers = (1, 5, 3, 8, 2)
min_num = min(numbers)
print(min_num)  # Output: 1
```

12. sum(): returns the sum of all the items in the tuple. Example

```
numbers = (1, 5, 3, 8, 2)
total = sum(numbers)
print(total)  # Output: 19
```

3.3 Frozenset

1. Definition: A frozen set is an immutable set in Python that cannot be modified once it is created.Example:

```
my_set = frozenset([1, 2, 3, 4])
print(my_set)  # Output: frozenset({1, 2, 3, 4})
```

2. Creation: A frozen set can be created using the built-in function `frozenset()` by passing an iterable object such as a list, set, or tuple to it. Example:

```
my_list = [1, 2, 3, 4]
my_set = frozenset(my_list)
print(my_set)  # Output: frozenset({1, 2, 3, 4})
```

3. Access: Elements of a frozen set can be accessed using a for loop or the `in` keyword to check if an element exists in the set.Example:

```
my_set = frozenset([1, 2, 3, 4])
for element in my_set:
    print(element)  # Output: 1 2 3 4
print(1 in my_set)  # Output: True
print(5 in my_set)  # Output: False
```

4. Methods: Since frozen sets are immutable, they have only two methods: `copy()` and `union()`. Example:

```
set1 = frozenset([1, 2, 3])
set2 = frozenset([3, 4, 5])
```

```
set3 = set1.union(set2)
print(set3)  # Output: frozenset({1, 2, 3, 4, 5})
```

5. Advantages: Frozen sets are useful when we need to use a set as a key in a dictionary or when we need to create a set of sets, since sets are mutable and cannot be hashed.

3.4 Set

A set is an unordered collection of unique elements in Python. It is similar to a list or a tuple, but unlike those data structures, sets cannot have duplicate values. Sets are represented by curly braces {} or by using the set() function.

```
# Creating an empty set
my_set = set()
```

```
# Creating a set with some initial values
my_set = {1, 2, 3}
```

Adding elements to a set:

```
my_set = {1, 2, 3}

# Adding a single element to the set
my_set.add(4)

# Adding multiple elements to the set
my_set.update([5, 6, 7])
```

Removing elements from a set:

```
my_set = {1, 2, 3, 4, 5}
```

```
# Removing a single element from the set
my_set.remove(4)

# Removing an element from the set, but ignoring it if it doesn't exist
my_set.discard(6)
# Removing and returning an arbitrary element from the set
my_set.pop()
```

Set operations:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Union of two sets
set3 = set1.union(set2)

# Intersection of two sets
set4 = set1.intersection(set2)

# Difference of two sets
set5 = set1.difference(set2)

# Symmetric difference of two sets
set6 = set1.symmetric_difference(set2)
```

Iterating over a set:

```
my_set = {1, 2, 3}
```

```
for num in my_set:
    print(num)
```

In Python, sets are iterable and can be looped over using a for loop. Overall, sets are a useful data structure in Python for keeping track of unique elements and performing set operations.

3.5 Dictionaries

Dictionaries are a type of collection in Python that store data in key-value pairs. Each key is unique and is used to access its corresponding value.

Creating and initializing a dictionary

We can create a dictionary by enclosing a comma-separated list of key-value pairs in curly braces **{}** or by using the built-in dict() constructor. Here's an example:

```
# Creating a dictionary
person = {'name': 'John', 'age': 30, 'city': 'New York'}
```

Accessing and modifying dictionary values using keys

We can access a value in a dictionary by using its key inside square brackets [] or by using the get() method. We can also modify the value of a key by assigning a new value to it. Here's an example:

```
# Accessing and modifying values in a dictionary
print(person['name'])      # Output: John
print(person.get('age'))   # Output: 30
person['city'] = 'Boston'
print(person)              # Output: {'name': 'John', 'age': 30, 'city': 'Boston'}
```

Checking if a key is in a dictionary

We can check if a key is present in a dictionary using the 'in' operator or the 'not in' operator. Here's an example:

```
# Checking if a key is in a dictionary
if 'name' in person:
    print('Name is present')
else:
    print('Name is not present')
```

Deleting values from a dictionary

We can delete a key-value pair from a dictionary using the del keyword or the pop() method. Here's an example:

```
# Deleting values from a dictionary
del person['age']
print(person)          # Output: {'name': 'John', 'city': 'Boston'}
person.pop('city')
print(person)          # Output: {'name': 'John'}
```

Dictionary Methods

Dictionaries in Python come with a number of built-in methods that allow you to perform various operations on them. This chapter covers some of the most commonly used methods of dictionaries:

Keys(), values(), and items() methods: These methods allow you to retrieve a list of all keys, values, or key-value pairs in a dictionary.

Copying and updating dictionaries: You can make a copy of an existing dictionary using the copy() method, or update it with the values from another dictionary using the update() method.

Pop() and popitem() methods: The pop() method removes a specified key-value pair from the dictionary and returns its value, while the popitem() method removes and returns an arbitrary key-value pair.

Clear() and setdefault() methods: The clear() method removes all items from a dictionary, while the setdefault() method sets a default value for a key that doesn't exist in the dictionary.

```
# Example of using the keys(), values(), and items() methods
my_dict = {'apple': 2, 'banana': 3, 'cherry': 4}
print(my_dict.keys())  # Output: dict_keys(['apple', 'banana', 'cherry'])
print(my_dict.values())  # Output: dict_values([2, 3, 4])
print(my_dict.items())  # Output: dict_items([('apple', 2), ('banana', 3), ('cherry', 4)])
# Example of copying and updating dictionaries
my_dict2 = my_dict.copy()
my_dict2.update({'orange': 5})
print(my_dict2)  # Output: {'apple': 2, 'banana': 3, 'cherry': 4, 'orange': 5}

# Example of using the pop() and popitem() methods
my_dict.pop('banana')
print(my_dict)  # Output: {'apple': 2, 'cherry': 4}
my_dict.popitem()
print(my_dict)  # Output: {'apple': 2}

# Example of using the clear() and setdefault() methods
my_dict.clear()
print(my_dict)  # Output: {}
print(my_dict.setdefault('apple', 2))  # Output: 2
print(my_dict)  # Output: {'apple': 2}
```

Advanced Dictionary Techniques

This covers advanced techniques for working with dictionaries, including merging dictionaries, sorting dictionaries, creating dictionaries with default values, handling missing keys, and preserving the order of dictionary keys.

Merging Dictionaries using update() Method

This method allows you to merge two or more dictionaries into a single dictionary.

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict1.update(dict2)
print(dict1)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Sorting Dictionaries using sorted() Method

This method allows you to sort a dictionary by its keys or values.

```
my_dict = {'apple': 3, 'banana': 1, 'pear': 2, 'orange': 4}
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1]))
print(sorted_dict)  # Output: {'banana': 1, 'pear': 2, 'apple': 3, 'orange': 4}
```

Creating Dictionaries with Default Values using defaultdict() Method

This method allows you to create a dictionary with default values for keys that haven't been set yet.

```
from collections import defaultdict
my_dict = defaultdict(int)
my_dict['a'] = 1
print(my_dict['b'])  # Output: 0
```

Handling Missing Keys using get() Method

This method allows you to handle missing keys without raising a KeyError.

```
my_dict = {'a': 1, 'b': 2}
print(my_dict.get('c', 0))  # Output: 0
```

Preserving Order of Dictionary Keys using OrderedDict() Method

This method allows you to create a dictionary that preserves the order in which the keys were inserted.

```
from collections import OrderedDict
my_dict = OrderedDict()
my_dict['a'] = 1
my_dict['b'] = 2
my_dict['c'] = 3
print(my_dict.keys())  # Output: odict_keys(['a', 'b', 'c']
```

3.6 Comprehensions

List Comprehension:

List comprehension is a concise way of creating a new list based on an existing list or iterable. It allows us to write a loop, a conditional statement, and an expression to create a new list in a single line of code. Here is an example:

```
# Create a new list with even numbers from 1 to 10
even_numbers = [x for x in range(1, 11) if x % 2 == 0]
print(even_numbers)  # Output: [2, 4, 6, 8, 10]
```

This code uses list comprehension to create a new list `even_numbers` with even numbers from 1 to 10. The expression `x for x in range(1, 11)` generates a sequence of numbers from 1 to 10, and the conditional statement `if x % 2 == 0` filters out the odd numbers. The result is a new list `[2, 4, 6, 8, 10]`.

Dictionary Comprehension:

Dictionary comprehension is similar to list comprehension, but it creates a new dictionary instead of a list. It allows us to create a dictionary by iterating over an iterable and generating key-value pairs based on a conditional statement. Here is an example:

```
# Create a dictionary of squares of numbers from 1 to 5
squares = {x: x**2 for x in range(1, 6)}
```

```
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

This code uses dictionary comprehension to create a new dictionary `squares` with keys as numbers from 1 to 5 and values as their squares. The expression `x: x**2` generates a key-value pair for each value of `x` in the sequence `range(1, 6)`. The result is a new dictionary `{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}`.

Set Comprehension:

Set comprehension is a concise way to create a new set from an iterable by applying a certain condition. It follows a syntax similar to list comprehension, but uses curly braces instead of square brackets.

```
# Create a set of even numbers from 1 to 10
even_set = {num for num in range(1, 11) if num % 2 == 0}
print(even_set)  # Output: {2, 4, 6, 8, 10}
```

Iterator:

An iterator is an object that can be iterated upon, meaning that it can be used in a loop. It must implement two methods: `__iter__()` and `__next__()`. The `__iter__()` method returns the iterator object itself, and the `__next__()` method returns the next value from the iterator.

```
# Create an iterator for a list
my_list = [1, 2, 3]
my_iter = iter(my_list)


# Iterate through the list using the iterator
```

```
print(next(my_iter)) # Output: 1
print(next(my_iter)) # Output: 2
print(next(my_iter)) # Output: 3
```

Generator Expressions:

Generator expressions are similar to list comprehensions and set comprehensions, but instead of creating a list or set, they create a generator object. This is useful when you don't want to create a large list in memory, but instead want to generate the values one at a time.

```
# Create a generator for even numbers from 1 to 10
even_gen = (num for num in range(1, 11) if num % 2 == 0)

# Print the values generated by the generator
for num in even_gen:
    print(num)  # Output: 2, 4, 6, 8, 10
```

Chapter 4

Loops & Control Statements

4.1 Control Statements

Control statements in Python are used to alter the flow of program execution based on certain conditions. There are three main types of control statements in Python: conditional statements, loop statements, and jump statements.

Conditional statements are used to execute a block of code only if a certain condition is met. The most commonly used conditional statement in Python is the if statement.

Loop statements are used to execute a block of code repeatedly. There are two main types of loops in Python: for loops and while loops.

Jump statements are used to alter the normal flow of program execution. The two main jump statements in Python are break and continue. The break statement is used to exit a loop early, while the continue statement is used to skip over a certain iteration of a loop.

Overall, control statements are essential tools for any programmer to have in their arsenal as they provide flexibility in how programs are executed and allow for more complex and efficient algorithms to be created.

Control statements in Python are used to alter the flow of program execution based on certain conditions. There are three main types of control statements in Python: conditional statements, loop statements, and jump statements.

Conditional statements are used to execute a block of code only if a certain condition is met. The most commonly used conditional statement in Python is the if statement.

Loop statements are used to execute a block of code repeatedly. There are two main types of loops in Python: for loops and while loops.

Jump statements are used to alter the normal flow of program execution. The two main jump statements in Python are break and continue. The break statement is used to exit a loop early, while the continue statement is used to skip over a certain iteration of a loop.

Conditional statements are an essential part of any programming language, including Python. They are used to perform different actions based on whether a condition is true or false. In Python, there are two types of conditional statements: **if and else.**

The basic syntax for an if statement is as follows:

```
if condition:
    # code to be executed if the condition is true
```

The code within the if statement must be indented to signify that it is part of the block of code to be executed if the condition is true. Here is an example:

```
x = 10
if x > 5:
   print("x is greater than 5")
```

In this example, the condition `x > 5` is true, so the code within the if statement is executed, which is to print the message "x is greater than 5".

The basic syntax for an if-else statement is as follows:

```
if condition:
    # code to be executed if the condition is true
else:
    # code to be executed if the condition is false
```

Here is an example:

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")
```

In this example, the condition `x > 5` is false, so the code within the else statement is executed, which is to print the message "x is less than or equal to 5".

Python also supports the use of elif statements, which allow for multiple conditions to be checked.

The basic syntax for an if-elif-else statement is as follows:

```
if condition1:
    # code to be executed if condition1 is true
elif condition2:
    # code to be executed if condition1 is false and condition2 is true
else:
    # code to be executed if both condition1 and condition2 are false
```

Here is an example:

```
x = 7
if x > 10:
    print("x is greater than 10")
elif x > 5:
```

```
    print("x is greater than 5 but less than or equal to 10")
else:
    print("x is less than or equal to 5")
```

In this example, the condition `x > 10` is false, so the code within the elif statement is checked. Since `x > 5` is true, the message "x is greater than 5 but less than or equal to 10" is printed.

Conditional statements can also be nested, allowing for even more complex logic to be implemented. Here is an example:

```
x = 8
if x > 5:
    if x > 10:
        print("x is greater than 10")
    else:
        print("x is greater than 5 but less than or equal to 10")
else:
    print("x is less than or equal to 5")
```

In this example, the outer if statement checks if `x > 5` is true. Since it is, the inner if statement is checked to see if `x > 10` is true. Since it is not, the message "x is greater than 5 but less than or equal to 10" is printed.

Conditional statements are powerful tools in Python that allow for complex logic to be implemented with ease. By understanding how they work and how to use them effectively, you can write more efficient and effective Python code.

One-liner if-else statement:

One-liner if-else statement is a shorthand way of writing a conditional statement in a single line of code. It is useful when we need to execute a single statement based on a condition.

```
# Print "Even" if a number is even, else "Odd"
num = 4
print("Even" if num % 2 == 0 else "Odd")  # Output: Even
```

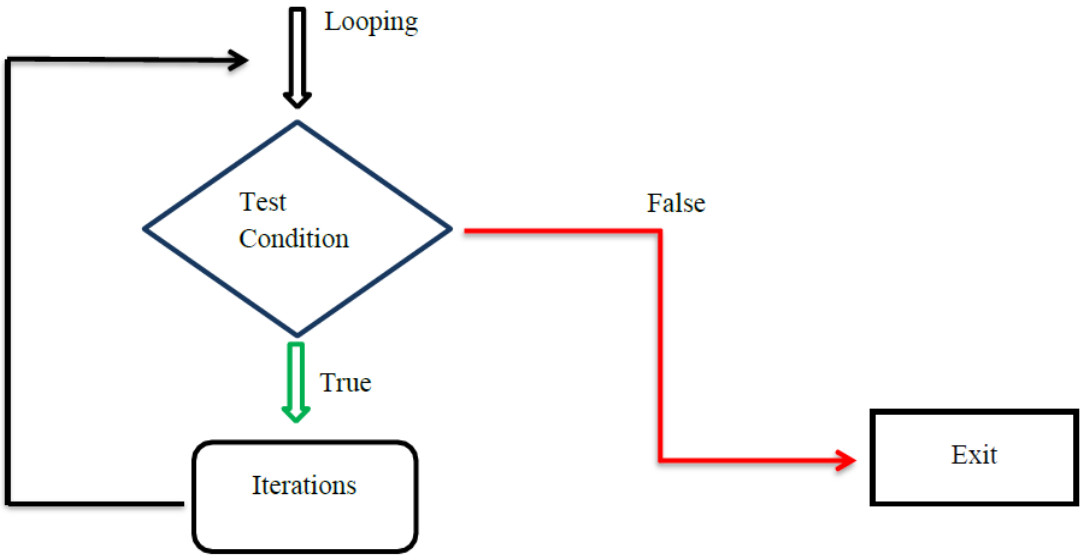
This code uses a one-liner if-else statement to print "Even" if the number `num` is even, else "Odd". The expression `"Even" if num % 2 == 0 else "Odd"` evaluates to "Even" if the condition `num % 2 == 0` is true, else "Odd". The result is the string "Even" printed to the console.

4.2 Loops

Loops in Python are used to execute a set of statements repeatedly. Python supports two types of loops: for loop and while loop.

For Loop

The for loop is used to iterate over a sequence (list, tuple, string, dictionary, set, etc.) and execute a block of code for each item in the sequence.The range() function can be used to generate a sequence of numbers that can be used to iterate over in a for loop. A for loop can be nested within another for loop to iterate over a nested sequence or perform a task for every combination of two sequences. The enumerate() function can be used with a for loop to iterate over the index and value of each item in a sequence.



Here's an example code snippet to illustrate the usage of these concepts:

```
# Iterating over a list
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)

# Output
apple
banana
cherry
```

```
# Using the range() function
for num in range(1, 6):
    print(num)

# Output
1
```

```
2
3
4
5
```

```
# Nested for loops
for x in range(1, 3):
    for y in range(1, 3):
        print(x, y)

# Output
1 1
1 2
2 1
2 2
```

```
# Using the enumerate() function
for index, fruit in enumerate(fruits):
    print(index, fruit)

# Output
0 apple
1 banana
2 cherry
```

```
# Example of for loop
fruits = ["apple", "banana", "cherry"]
```

```
for fruit in fruits:
    print(fruit)


# Output:
apple
banana
cherry
```

While Loop

The while loop is used to execute a block of code repeatedly as long as a certain condition is true. While loop is a pretest loop where the loop's body is executed only if the condition is true. It keeps looping until the condition becomes false or until a break statement is encountered.

```
# Example of while loop
i = 1
while i <= 5:
    print(i)
    i += 1


# Output:
1
2
3
4
5
```

Loop Control Statements

Python provides two loop control statements: break and continue.

- **break** statement is used to terminate the loop when a certain condition is met.

```
# Example of break statement
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    if fruit == "banana":
        break
    print(fruit)
# Output: apple
```

- **continue** statement is used to skip the current iteration of the loop when a certain condition is met.

```
# Example of continue statement
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    if fruit == "banana":
        continue
    print(fruit)

# Output:
apple
cherry
```

Nested Loops

Python allows you to use one or more loops inside another loop. These are called nested loops.

```
# Example of nested loops
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
Output:


Copy code
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Loops are an essential part of programming in Python. They allow you to execute a block of code repeatedly and control the flow of your program.

Chapter 5

Functions, Classes and Exception handling

5.1 Functions

Functions in Python are blocks of reusable code that perform a specific task. They help in modularizing the code and making it more organized. Functions can take parameters as inputs and return values as outputs.

There are two types of functions:

1. Builtin Function
2. User defined Function

Builtin Function

`**map()**` takes two arguments: a function and an iterable, and returns a new iterable with the function applied to each item in the iterable.

```
# Create a list of numbers
numbers = [1, 2, 3, 4, 5]


# Create a new list with each number doubled
doubled_numbers = list(map(lambda x: x * 2, numbers))
print(doubled_numbers)  # Output: [2, 4, 6, 8, 10]
```

`**filter()**` takes two arguments: a function and an iterable, and returns a new iterable with only the items from the iterable that satisfy the condition in the function.

```
# Create a list of numbers
numbers = [1, 2, 3, 4, 5]
```

```
# Create a new list with only the even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

`**reduce()**` takes two arguments: a function and an iterable, and returns a single value that is the result of applying the function to each item in the iterable, in turn.

```
from functools import reduce

# Create a list of numbers
numbers = [1, 2, 3, 4, 5]

# Calculate the sum of the numbers
sum = reduce(lambda x, y: x + y, numbers)
print(sum)  # Output: 15
```

Some of the most commonly used built-in functions in Python along with their descriptions.

- **abs():** Returns the absolute value of a number.
- **all():** Returns True if all elements in an iterable are true.
- **any():** Returns True if any element in an iterable is true.
- **bin():** Converts an integer to a binary string.
- **bool():** Converts a value to a boolean.
- **callable():** Returns True if an object is callable.
- **chr():** Returns a character that corresponds to an ASCII value.
- **dir():** Returns a list of valid attributes for an object.

- **divmod():** Returns the quotient and remainder of two numbers.
- **enumerate():** Returns an iterator that yields tuples containing an index and an item from an iterable.
- **filter():** Filters an iterable by removing elements that don't satisfy a certain condition.
- **float():** Converts a string or integer to a float.
- **format():** Formats a string.
- **help():** Displays information about an object.
- **input():** Reads a line of input from the user.
- **int():** Converts a string or float to an integer.
- **isinstance():** Returns True if an object is an instance of a specified class.
- **len():** Returns the length of an object.
- **list():** Converts an iterable to a list.
- **map():** Applies a function to each item in an iterable.
- **max():** Returns the largest item in an iterable.
- **min():** Returns the smallest item in an iterable.
- **next():** Returns the next item in an iterator.
- **open():** Opens a file and returns a file object.
- **ord():** Returns an ASCII value that corresponds to a character.
- **print():** Prints one or more values to the console.
- **range():** Returns an iterable of numbers within a specified range.
- **repr():** Returns a string representation of an object.
- **reversed():** Returns a reversed iterator of an iterable.
- **round():** Rounds a number to a specified number of decimal places.
- **set():** Converts an iterable to a set.
- **sorted():** Returns a sorted list of an iterable.
- **str():** Converts a value to a string.
- **sum():** Returns the sum of all elements in an iterable.
- **tuple():** Converts an iterable to a tuple.

- **type():** Returns the type of an object.
- **zip():** Returns an iterator that aggregates elements from two or more iterables.

These functions are very useful and can be used in various ways to simplify and optimize your code.We can find more functions in Official Documentation.

User Defined Functions

Basic Syntax

```
def function_name(parameter1, parameter2, ...):
    # code block
    return output_value
```

Here, `def` is a keyword that is used to define a function. `function_name` is the name of the function that you want to define. Inside the parentheses, you can list the parameters that the function takes. These parameters are optional, and you can have a function that takes no parameters at all. The code block inside the function is indented, and it contains the instructions that the function executes. Finally, the `return` statement is used to return a value from the function.

Function Parameters

Functions can take parameters as inputs. There are two types of parameters:

1. **Positional parameters:** These are parameters that are matched by position. They are passed to the function in the order that they are defined.

2. **Keyword parameters:** These are parameters that are matched by name. They are passed to the function as key-value pairs.

```
def greet(name, message):
    print(f"{message}, {name}!")

# positional arguments
greet("John", "Hello")  # output: Hello, John!

# keyword arguments
greet(message="Hi", name="Jane")  # output: Hi, Jane!
```

Default Parameters

You can define default values for parameters in a function. If the value of a parameter is not provided when the function is called, it will take on the default value.

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

# using default parameter value
greet("John")  # output: Hello, John!
# overriding default parameter value
greet("Jane", "Hi")  # output: Hi, Jane!
```

Return Values

Functions can return values using the `return` keyword. If a function does not return a value, it returns `None` by default.

```
def add(a, b):
```

```
    return a + b


result = add(3, 4)
print(result)  # output: 7
```

Lambda Functions

Lambda functions are anonymous functions that can be defined in a single line of code. They are useful for simple operations where defining a named function would be overkill.

```
double = lambda x: x * 2
result = double(3)
print(result)  # output: 6
```

Functions as Objects

In Python, functions are first-class objects, which means they can be treated like any other object. They can be passed as arguments to other functions, returned as values from functions, and assigned to variables.

```
def add(a, b):
    return a + b


def subtract(a, b):
    return a - b


def apply(func, a, b):
    return func(a, b)
```

```
result1 = apply(add, 3, 4)  # output: 7
result2 = apply(subtract, 3, 4)  # output: -1
```

Decorators

Decorators are a feature of Python that allow you to modify the behavior of a function or class by wrapping it with another function. The decorator function takes the original function as an argument and returns a new function that adds some additional functionality. Decorators are useful for adding functionality to existing code without modifying the original code.

In this example, the `log` function is a decorator that takes a function `func` as an argument and returns a new function `wrapper` that logs the arguments and return value of `func`. The `wrapper` function takes any number of arguments and keyword arguments using the `*args` and `**kwargs` syntax. Finally, the `wrapper` function returns the result of the original function.

The `@log` syntax is a shorthand way of applying the `log` decorator to the `add` function. When the `add` function is called, it is actually the `wrapper` function that is executed, which logs the arguments and return value of `add`.

```
def log(func):
   def wrapper(*args, **kwargs):
      print(f"Calling function {func.__name__} with arguments {args} and {kwargs}")
      result = func(*args, **kwargs)
      print(f"Function {func.__name__} returned {result}")
      return result
```

```
    return wrapper


@log
def add(a, b):
    return a + b


result = add(1, 2)


# Output
Calling function add with arguments (1, 2) and {}
Function add returned 3
```

Decorators can be chained together to apply multiple decorators to a single function, and they can be used to implement many different types of functionality, such as caching, timing, authentication, and more.

In Python, `*args` and `**kwargs` are special syntax used in function definitions that allow you to pass a variable number of arguments to a function.

`***args**` is used to pass a variable number of non-keyword arguments to a function. It allows you to pass any number of arguments to a function as a tuple.

```
def my_func(*args):
    for arg in args:
        print(arg)


my_func(1, 2, 3, "hello")
```

```
# Output:
1
2
3
hello
```

`**kwargs` is used to pass a variable number of keyword arguments to a function. It allows you to pass any number of keyword arguments to a function as a dictionary.

```
def my_func(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

my_func(name="Alice", age=25, city="New York")

# Output:
name: Alice
age: 25
city: New York
```

You can also use `*args` and `**kwargs` together in a function definition to accept both non-keyword and keyword arguments. The `*args` parameter must come before the `**kwargs` parameter.

```
def my_func(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(f"{key}: {value}")

my_func(1, 2, 3, name="Alice", age=25, city="New York")

# Output:
1
2
3
name: Alice
age: 25
city: New York
```

In Python, a variable's scope determines where it can be accessed or modified in a program. There are two types of variable scopes: global and local.

Global variables are defined outside of any function or class, and they can be accessed from anywhere in the program. Any variable defined inside a function or class has local scope and can only be accessed within that function or class.

Here is an example that demonstrates the difference between global and local variables:

```
# Global variable
name = "John"

def greet():
    # Local variable
    name = "Jane"
    print("Hello, " + name)

greet() # Output: Hello, Jane
print("Hi, " + name) # Output: Hi, John
```

In the example above, `name` is a global variable defined outside of the `greet` function. Inside the function, we define a new variable with the same name, but it has local scope and only exists within the function. When we call the `greet` function, it prints "Hello, Jane" because it uses the local `name` variable. Outside the function, we print "Hi, John" because we are accessing the global `name` variable.

If we want to modify a global variable inside a function, we can use the `global` keyword to tell Python that we are referring to the global variable, not a local variable with the same name.

```
# Global variable
count = 0
```

```
def increment():
    # Use the global keyword to modify the global count variable
    global count
    count += 1


increment()
print(count) # Output: 1
```

In this example, we define a global variable `count` and a function `increment` that modifies the global `count` variable using the `global` keyword. When we call the `increment` function, it increments the global `count` variable and prints 1.

5.2 Class

In Python, a class is a fundamental concept that allows you to create objects with attributes and methods. A class defines a set of attributes that an object can have, and methods that an object can perform. You can think of a class as a blueprint that specifies the attributes and behavior of an object.

An instance of a class is a specific object that is created from the class, and it can have its own set of attributes and call its own set of methods. This makes it easy to create multiple objects that have similar attributes and behavior.

One of the powerful features of classes in Python is inheritance. With inheritance, you can create a new class from an existing class and inherit all its attributes and methods. This allows you to reuse code and avoid duplicating functionality.

Polymorphism is another important concept in object-oriented programming, and it refers to the ability of objects to take on different forms. In

Python, this is achieved through method overriding and method overloading. Method overriding allows a subclass to provide a different implementation of a method that is already defined in the parent class, while method overloading allows you to define multiple methods with the same name but different parameters.

Overall, understanding classes and object-oriented programming is crucial for building complex and modular applications in Python.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def get_make(self):
        return self.make

    def get_model(self):
        return self.model

    def get_year(self):
        return self.year

my_car = Car("Ford", "Mustang", 2022)
print(my_car.get_make())  # Output: Ford
print(my_car.get_model())  # Output: Mustang
```

```
print(my_car.get_year())  # Output: 2022
```

In this example, we have defined a class called "Car" with three attributes (make, model, and year) and three methods (get_make, get_model, and get_year). We then create an instance of this class called "my_car" with the make "Ford", the model "Mustang", and the year "2022". We can then call the methods on this instance to get the values of the attributes.

Constructor

In Python, a constructor is a special method that is called when an object of a class is created. It is used to initialize the attributes of the object. The constructor method is always named `__init__()` and takes `self` as its first parameter, which refers to the instance of the class that is being created.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

In this example, the `Person` class has a constructor that takes two parameters, `name` and `age`. These parameters are used to set the `name` and `age` attributes of the object.

When an object of the `Person` class is created, the constructor is called automatically.

```
person = Person("John", 30)
```

In this example, the constructor is called with the values `"John"` and `30`, which are used to set the `name` and `age` attributes of the `person` object.

Constructors are useful because they ensure that objects are initialized properly when they are created. They can also perform any additional setup or validation that is needed before the object is ready to be used.

In Python, the "@" symbol is used as a decorator to modify the behavior of a function or class. Here are some of the commonly used "@ methods" in Python:

1. **@property:** This decorator is used to define a method as a property of a class. It allows you to get and set the value of an attribute as if it were a regular attribute, but with custom getter and setter methods.

```
class MyClass:
    def __init__(self):
        self._x = 0
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value

obj = MyClass()
obj.x = 10
print(obj.x) # Output: 10
```

2. **@classmethod:** This decorator is used to define a class method. Class methods are methods that are bound to the class and not the instance of the class. They can be called on the class itself, rather than on an instance of the class.

```
class MyClass:
    count = 0

    def __init__(self):
        MyClass.count += 1

    @classmethod
    def get_count(cls):
        return cls.count

obj1 = MyClass()
obj2 = MyClass()
print(MyClass.get_count()) # Output: 2
```

cls is a conventional name used for the first parameter of the method. It represents the class itself, rather than an instance of the class.

3. **@staticmethod:** This decorator is used to define a static method. Static methods are methods that belong to the class and not the instance of the class. They do not operate on the instance of the class and do not have access to instance-specific data.

```
class MyClass:
    @staticmethod
```

```
    def say_hello():
        print("Hello, World!")

MyClass.say_hello() # Output: Hello, World!
```

4. **@abstractmethod:** This decorator is used to define an abstract method. Abstract methods are methods that are declared in a base class, but do not have an implementation. They must be implemented in any concrete subclasses.

```
from abc import ABC, abstractmethod
class MyAbstractClass(ABC):
    @abstractmethod
    def do_something(self):
        pass

class MyClass(MyAbstractClass):
    def do_something(self):
        print("Doing something...")

obj = MyClass()
obj.do_something() # Output: Doing something…
```

5. **@asyncio.coroutine:** This decorator is used to define a coroutine function. Coroutines are a way to write asynchronous code in Python. They allow you to write code that can be paused and resumed at any point, without blocking the main thread.

```
import asyncio


@asyncio.coroutine
def my_coroutine():
    print('Coroutine started')
    yield from asyncio.sleep(1)
    print('Coroutine ended')


loop = asyncio.get_event_loop()
loop.run_until_complete(my_coroutine())
loop.close()
```

In this example, my_coroutine() is marked with the @asyncio.coroutine decorator, which indicates that it is a coroutine function. The yield from statement is used to delegate to another coroutine function, asyncio.sleep(), which suspends the execution of the coroutine for one second. The asyncio.get_event_loop() method is used to get the event loop, and the loop.run_until_complete() method is used to run the coroutine until it completes. Finally, the loop.close() method is used to close the event loop.

These are just a few examples of the many decorators available in Python. Decorators can be a powerful tool for modifying the behavior of functions and classes, and can be used to implement many different programming patterns.

Double Underscore methods

Double underscore methods in Python are also known as magic methods or dunder methods. These methods have special meaning in Python, and they are

used to define the behavior of built-in functions and operators. Here are some commonly used double underscore methods:

Method	Description
__init__(self, ...)	Initializes an object of the class
__str__(self)	Returns a string representation of the object
__repr__(self)	Returns a string representation of the object that can be used to recreate the object
__len__(self)	Returns the length of the object
__getitem__(self, key)	Returns the value associated with the given key
__setitem__(self, key, value)	Sets the value associated with the given key
__delitem__(self, key)	Deletes the item associated with the given key
__iter__(self)	Returns an iterator object
__next__(self)	Returns the next value from the iterator
__enter__(self)	Called when entering a with statement
__exit__(self,exc_type, exc_value, traceback)	Called when exiting a with statement

```
class MyList:
    def __init__(self, *args):
        self.data = list(args)

    def __str__(self):
        return str(self.data)

    def __getitem__(self, i):
        return self.data[i]
```

```
    def __setitem__(self, i, value):
        self.data[i] = value

    def __len__(self):
        return len(self.data)

    def __iter__(self):
        return iter(self.data)

my_list = MyList(1, 2, 3)
print(my_list)  # Output: [1, 2, 3]
print(my_list[1])  # Output: 2
my_list[1] = 4
print(my_list)  # Output: [1, 4, 3]
print(len(my_list))  # Output: 3
for item in my_list:
    print(item)  # Output: 1, 4, 3
```

5.3 Exception handling

Exception handling is a way to handle errors that occur during the execution of a program. In Python, exception handling is done using the try-except block. The try block contains the code that may raise an exception, and the except block contains the code that is executed when an exception is raised.

```
try:
    # Code that may raise an exception
    a = 1 / 0
except ZeroDivisionError:
    # Code that handles the exception
    print("Cannot divide by zero")
```

In this example, the code inside the try block may raise a `ZeroDivisionError` exception because we are trying to divide a number by zero. If this exception is raised, the code inside the except block will be executed, which simply prints a message saying that we cannot divide by zero.

Python also provides a way to handle multiple exceptions using multiple except blocks:

```
try:
    # Code that may raise an exception
    a = int("hello")
except ZeroDivisionError:
    # Code that handles ZeroDivisionError
    print("Cannot divide by zero")
except ValueError:
    # Code that handles ValueError
    print("Cannot convert string to integer")
```

In this example, we are trying to convert the string "hello" to an integer using the `int()` function. This will raise a `ValueError` exception because the string cannot be converted to an integer. We have two except blocks, one for

handling `ZeroDivisionError` and one for handling `ValueError`. If the `ValueError` exception is raised, the code inside the second except block will be executed, which prints a message saying that we cannot convert the string to an integer.

Finally, we can also use a `finally` block to execute code that should always be executed, whether or not an exception is raised:

```
try:
    # Code that may raise an exception
    f = open("myfile.txt")
    lines = f.readlines()
except IOError:
    # Code that handles IOError
    print("Could not read file")
finally:
    # Code that is always executed
    f.close()
```

In this example, we are trying to read the contents of a file named "myfile.txt". This may raise an `IOError` exception if the file cannot be opened. If this exception is raised, the code inside the except block will be executed, which prints a message saying that we could not read the file. The code inside the finally block will always be executed, regardless of whether an exception was raised or not. In this case, we are closing the file handle to make sure that we release any system resources that were used to open the file.

Exception	Description
Exception	Base class for all exceptions.
TypeError	Raised when an operation or function is applied to an object of inappropriate type.
ValueError	Raised when an operation or function receives an argument of correct type but inappropriate value.
AttributeError	Raised when an attribute reference or assignment fails.
NameError	Raised when a local or global name is not found.
IndexError	Raised when a sequence index is out of range.
KeyError	Raised when a dictionary key is not found.
ZeroDivisionError	Raised when the second operand of a division or modulo operation is zero.
FileNotFoundError	Raised when a file or directory is requested but doesn't exist.
IOError	Raised when an I/O operation fails.
NotImplementedError	Raised when an abstract method is called or when an operation is not implemented.
KeyboardInterrupt	Raised when the user presses Ctrl+C.
SystemExit	Raised when the Python interpreter is exited.

Note that this is not an exhaustive list of all possible exceptions, but covers some of the most common ones.

Chapter 6

File I/O,Regex and Recursion

6.1 File I/O

In Python, file input/output (I/O) refers to the operations performed on files, including reading from and writing to files. Here is a brief overview of file I/O in Python:

Opening a file: Before you can read from or write to a file, you need to open it using the built-in `open()` function. This function takes two arguments: the name of the file, and the mode in which to open the file (read mode, write mode, append mode, etc.).

Reading from a file: Once you have opened a file in read mode, you can read from it using methods like `read()`, `readline()`, and `readlines()`.

Writing to a file: Once you have opened a file in write mode, you can write to it using methods like `write()` and `writelines()`.

Closing a file: After you have finished reading from or writing to a file, you should close it using the `close()` method.

```
# Open a file in read mode
file = open("example.txt", "r")

# Read the contents of the file
contents = file.read()
print(contents)

# Close the file
file.close()

# Open a file in write mode
```

```
file = open("example.txt", "w")


# Write some text to the file
file.write("Hello, world!")


# Close the file
file.close()
```

In this example, we first open a file called "example.txt" in read mode and read its contents using the `read()` method. Then we close the file. Next, we open the same file in write mode and write the string "Hello, world!" to it using the `write()` method. Finally, we close the file again.

Note that it's always a good practice to close a file when you are done with it to avoid issues like resource leaks and data corruption.

In Python, `**with open**` is a common way to open and read/write files. It ensures that the file is properly closed after it has been used, even if an exception is raised. Here's an example of how to use `with open`:

```
with open('example.txt', 'r') as f:
    content = f.read()
    print(content)
```

In this example, the `open` function is used to open the file `example.txt` in read mode (`'r'`). The file object is assigned to the variable `f`. The `with` statement ensures that the file is properly closed when the block is exited.

The `read` method is used to read the contents of the file and store it in the variable `content`. The contents of the file are then printed to the console.

Similarly, you can use `with open` to write to a file:

```
with open('example.txt', 'w') as f:
    f.write('Hello, world!')
```

In this example, the file `example.txt` is opened in write mode (`'w'`). The file object is assigned to the variable `f`. The `with` statement ensures that the file is properly closed when the block is exited. The `write` method is used to write the string `'Hello, world!'` to the file.

Method	Description
file = open(filename, mode)	Open a file with the specified name and mode. Returns a file object.
file.close()	Close the file object.
file.read()	Read the entire contents of the file as a string.
file.read(size)	Read up to size bytes from the file as a string.
file.readline()	Read the next line from the file as a string.
file.readlines()	Read all the lines of the file into a list of strings.
file.write(string)	Write the string to the file.
file.writelines(list)	Write the list of strings to the file.
file.tell()	Return the current position in the file as an integer.
file.seek(offset, whence)	Change the current position in the file. whence can be 0 (beginning of file), 1 (current position), or 2 (end of file).
os.remove(filename)	Delete the file with the specified name.
os.rename(oldname, newname)	Rename the file with the old name to the new name.

6.2 Regex

Regular expressions, also known as regex or regexp, are a powerful tool for searching, matching, and manipulating text in computer programming languages. A regular expression is a sequence of characters that defines a search pattern.

Regex is used in many programming languages including Python, Java, JavaScript, Perl, Ruby, and more. It is especially useful for data validation, text parsing, and data extraction tasks.

In Python, the `re` module provides support for regular expressions. Here are some of the most commonly used functions and methods for working with regular expressions in Python:

1. **`re.search(pattern, string, flags=0)`:** Searches for the first occurrence of the specified pattern in the given string and returns a match object. If no match is found, return `None`.

2. **`re.match(pattern, string, flags=0)`:** Searches for the pattern at the beginning of the string and returns a match object. If no match is found, return `None`.

3. `re.findall(pattern, string, flags=0)`: Searches for all occurrences of the specified pattern in the given string and returns a list of all matches.

4. `re.sub(pattern, repl, string, count=0, flags=0)`: Replaces all occurrences of the specified pattern in the string with the replacement string `repl`. The `count` parameter can be used to limit the number of replacements.

5. `re.compile(pattern, flags=0)`: Compiles the specified regular expression pattern into a regular expression object, which can be used for matching, searching, and replacing text.

6. `match.group([group1, ...])`: Returns one or more subgroups of the match. If no arguments are provided, return the entire match.

7. `match.start([group])`: Returns the starting position of the match.

8. `match.end([group])`: Returns the ending position of the match.

9. **`match.span([group])`:** Returns a tuple containing the start and end positions of the match.

```
import re

text = "The quick brown fox jumps over the lazy dog"
pattern = r"fox"

# search for pattern in text
match = re.search(pattern, text)
if match:
    print("Found:", match.group())
else:
    print("Not found")

# replace pattern in text
new_text = re.sub(pattern, "cat", text)
print(new_text)
```

```
# find all matches of pattern in text
matches = re.findall(pattern, text)
print(matches)


Output:
Found: fox
The quick brown cat jumps over the lazy dog
['fox']
```

In this example, we use the `re.search` function to search for the pattern "fox" in the text "The quick brown fox jumps over the lazy dog". Since the pattern is found, we print "Found: fox".

Next, we use the `re.sub` function to replace all occurrences of the pattern "fox" with "cat" in the text. The resulting text is printed.

Finally, we use the `re.findall` function to find all occurrences of the pattern "fox" in the text and print the resulting list.

Pattern	Description
.	Matches any character except newline
\d	Matches any digit character
\D	Matches any non-digit character
\w	Matches any word character (letter, digit, underscore)
\W	Matches any non-word character
\s	Matches any whitespace character (space, tab, newline)
\S	Matches any non-whitespace character

[ ]	Matches any character in the specified set
[^]	Matches any character not in the specified set
^	Matches the beginning of a line
$	Matches the end of a line
*	Matches zero or more occurrences of the preceding character
+	Matches one or more occurrences of the preceding character
?	Matches zero or one occurrences of the preceding character
{n}	Matches exactly n occurrences of the preceding character
{n,}	Matches at least n occurrences of the preceding character
{n,m}	Matches between n and m occurrences of the preceding character

Here are two examples of pattern matching using regular expressions in Python:

1. Simple Pattern Matching: Matching Email Addresses

```
import re

# Define a regular expression pattern for matching email addresses
pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

# Sample text to search for email addresses
text = "Email me at john@example.com or jane@example.co.uk"

# Use the search() method from the re module to find matches
matches = re.search(pattern, text)
# Print the match object
```

```
print(matches.group())


Output: `john@example.com`
```

In this example, we define a regular expression pattern to match email addresses. The pattern uses several pattern characters to match the username, domain, and top-level domain of an email address. We then use the search() method from the re module to search for matches in a sample text string.

2. Complex Pattern Matching: Extracting Phone Numbers from a Text File

```
import re

# Define a regular expression pattern for matching phone numbers
pattern = r'\b(\d{3}[-.]|\(\d{3}\) )?\d{3}[-.]\d{4}\b'

# Read the contents of a text file
with open('sample.txt', 'r') as f:
    text = f.read()

# Use the findall() method from the re module to extract phone numbers
matches = re.findall(pattern, text)

# Print the list of phone numbers
print(matches)

Output: `['123-456-7890', '(123) 456-7890']`
```

In this example, we define a regular expression pattern to match phone numbers in various formats. We then read the contents of a text file and use the findall() method from the re module to extract all instances of phone numbers that match the pattern. The resulting list of phone numbers is then printed to the console.

6.3 Recursion

Recursion is a technique in which a function calls itself. It is a powerful and elegant way to solve problems that can be broken down into smaller, similar problems.

Example of a simple recursive function that calculates the factorial of a number:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

In this function, if the value of `n` is zero, the function returns 1. Otherwise, it returns `n` times the factorial of `n-1`.

Example of a more complex recursive function that searches for a target value in a nested list:

```
def search_nested_list(target, nested_list):
    for element in nested_list:
        if isinstance(element, list):
```

```
        result = search_nested_list(target, element)
        if result is not None:
            return result
    elif element == target:
        return element
return None
```

In this function, the `search_nested_list` function takes a target value and a nested list as input. It then iterates over each element in the list, and if an element is itself a list, the function calls itself recursively to search that nested list. If the target value is found, it is returned. If the entire list is searched and the target value is not found, the function returns `None`.

Chapter 7

Modules

7.1 OS Module

The `os` module in Python provides a way of interacting with the operating system. It includes various functions for accessing the file system, managing processes, and performing system-related tasks.

Here are some commonly used functions in the `os` module:

1. `os.name`: returns the name of the operating system (e.g. 'posix' for Unix/Linux, 'nt' for Windows)

2. `os.getcwd()`: returns the current working directory as a string

3. `os.chdir(path)`: changes the current working directory to the specified path

4. `os.listdir(path='.')`: returns a list of files and directories in the specified directory

5. `os.mkdir(path)`: creates a new directory at the specified path

6. `os.remove(path)`: removes the file at the specified path

7. `os.rmdir(path)`: removes the directory at the specified path

8. `os.path.join(path1, path2, ...)`: joins two or more paths together to form a complete path

9. `os.path.exists(path)`: returns True if the path exists, False otherwise

```
import os

# Get the current working directory
cwd = os.getcwd()
print("Current working directory:", cwd)

# Create a new directory
```

```
new_dir = os.path.join(cwd, "new_directory")
os.mkdir(new_dir)

# Check if the new directory exists
if os.path.exists(new_dir):
    print("New directory created successfully")

# Change the working directory
os.chdir(new_dir)
print("New working directory:", os.getcwd())

# Create a new file in the new directory
new_file = os.path.join(os.getcwd(), "new_file.txt")
with open(new_file, "w") as f:
    f.write("This is a new file")

# List the contents of the directory
contents = os.listdir()
print("Contents of directory:", contents)

# Remove the new file and directory
os.remove(new_file)
os.chdir(cwd)
os.rmdir(new_dir)
```

7.2 Sys Module

Sure, I can provide more details about the sys module. The sys module provides access to some variables and functions used or maintained by the Python interpreter.

1. **sys.argv:** This is a list in Python, which contains the command-line arguments passed to the script. With this, you can get the name of the script, the number of arguments passed and their values.
2. **sys.exit():** This function is used to exit the Python interpreter. It takes an optional argument which is an integer indicating the exit status. By convention, 0 means success, and any other value means some kind of failure.
3. **sys.path:** This is a list of strings that specifies the search path for modules. When a module is imported, Python searches for it in each directory listed in sys.path, in the order they are listed.
4. **sys.stdin, sys.stdout, and sys.stderr:** These are file-like objects that represent standard input, standard output, and standard error, respectively.
5. **sys.platform:** This returns a string that identifies the platform on which Python is running, such as 'linux', 'win32', 'darwin' (for macOS), etc.

Some of the features of the **sys** module:

```
import sys


# Print the command-line arguments passed to the script
print("Command-line arguments:", sys.argv)


# Get the name of the script
```

```
print("Script name:", sys.argv[0])

# Get the number of arguments passed
print("Number of arguments:", len(sys.argv) - 1)

# Get the values of the arguments passed
print("Argument values:", sys.argv[1:])

# Exit the script with a non-zero status
sys.exit(1)

# Print the search path for modules
print("Module search path:", sys.path)

# Print the platform on which Python is running
print("Platform:", sys.platform)

# Print some information about standard input
print("Standard input:", sys.stdin)

# Print some information about standard output
print("Standard output:", sys.stdout)

# Print some information about standard error
print("Standard error:", sys.stderr)
```

Note that in this example, the code after `sys.exit(1)` is not executed because the script is terminated when `sys.exit()` is called. Also, the output of the script may vary depending on the platform and the arguments passed to it.

7.3 Turtle Module

The turtle module in Python is a standard library module that provides a simple way to create graphics and animations using a turtle-like object. It is a beginner-friendly module that helps in introducing concepts of programming and algorithms.

1. **`turtle.forward(distance)`** - This function moves the turtle forward by a specified distance in the direction it's facing. For example, `turtle.forward(100)` will move the turtle forward by 100 pixels.

2. **`turtle.backward(distance)`** - This function moves the turtle backward by a specified distance in the opposite direction it's facing. For example, `turtle.backward(50)` will move the turtle backward by 50 pixels.

3. **`turtle.right(angle)`** - This function turns the turtle right by a specified angle. For example, `turtle.right(90)` will turn the turtle 90 degrees to the right.

4. **`turtle.left(angle)`** - This function turns the turtle left by a specified angle. For example, `turtle.left(45)` will turn the turtle 45 degrees to the left.

5. **`turtle.penup()`** - This function lifts the turtle's pen off the drawing surface, so it won't draw when moving. For example, `turtle.penup()` will make the turtle stop drawing.

6. **`turtle.pendown()`** - This function puts the turtle's pen back on the drawing surface, so it will draw when moving. For example, `turtle.pendown()` will make the turtle start drawing again.

7. **`turtle.pencolor(color)`** - This function sets the color of the turtle's pen. The color can be specified using a string representing a color name or an RGB tuple. For example, `turtle.pencolor('red')` or `turtle.pencolor((255, 0, 0))` will set the pen color to red.

8. **`turtle.fillcolor(color)`** - This function sets the fill color for the turtle's shape. The color can be specified using a string representing a color name or an RGB tuple. For example, `turtle.fillcolor('blue')` or `turtle.fillcolor((0, 0, 255))` will set the fill color to blue.

9. **`turtle.begin_fill()`** and **`turtle.end_fill()`** - These functions are used to fill in a shape drawn by the turtle with the fill color. **`turtle.begin_fill()`** starts the fill and `turtle.end_fill()` ends it.

```
turtle.begin_fill()
turtle.circle(50)
turtle.end_fill()
```

will draw a circle with a radius of 50 pixels and fill it with the current fill color.

10. **`turtle.circle(radius, extent=None)`** - This function draws a circle with the specified radius. The extent parameter, if given, specifies the angle of the arc to be drawn (in degrees). For example, `turtle.circle(50)` will draw a full circle with a radius of 50 pixels, while `turtle.circle(50, 180)` will draw a half-circle with the same radius.

These are just a few of the many functions available in the turtle module. By using these functions in combination with loops and conditional statements, it's possible to create more complex drawings and animations.

Here's an example program that uses some of these turtle functions to draw a simple flower:

```
import turtle


turtle.speed(0)


for i in range(6):
    turtle.penup()
    turtle.goto(0, 0)
    turtle.pendown()
    turtle.right(60 * i)
    for j in range(3):
        turtle.forward(50)
        turtle.right(120)
    turtle.right(60)


turtle.exitonclick()
```

7.4 MySql database

The MySQL module in Python is used to interact with a MySQL database. It provides an interface for connecting to a MySQL server, executing queries, and retrieving results. Here's a brief overview of how to use the MySQL module in Python:

1. Install the MySQL module:

```
pip install mysql-connector-python
```

2. Import the module :

```
import mysql.connector
```

3. Connect to a MySQL server :

```
mydb = mysql.connector.connect(
  host="localhost",
  user="username",
  password="password",
  database="database_name"
)
```

4. Create a cursor object :

```
mycursor = mydb.cursor()
```

5. Execute a query :

```
mycursor.execute("SELECT * FROM table_name")
```

6. Fetch the results :

```
result = mycursor.fetchall()
```

7. Close the connection :

```
mydb.close()
```

This is just a basic example of how to use the MySQL module. There are many more advanced features available, such as prepared statements,

transactions, and connection pooling. It's recommended to refer to the official documentation for more information and examples.

Create :

To execute MySQL queries in Python, we need to use a MySQL connector. The `mysql-connector-python` is one such connector that allows Python to connect to a MySQL database.

```
import mysql.connector

# connect to the database
db = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="yourdatabase"  )

# create a cursor object
cursor = db.cursor()

# define the SQL query to create a table
sql = "CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255), address VARCHAR(255))"

# execute the query
cursor.execute(sql)
```

```
# commit the changes to the database
db.commit()

# print a success message
print("Table created successfully.")
```

In the above code, we first establish a connection to the MySQL database using the `mysql.connector.connect()` function and create a cursor object using the `cursor()` method.

Next, we define our SQL query to create a table and execute it using the `execute()` method of the cursor object. We then commit the changes to the database using the `commit()` method and print a success message.

Insert Data

To insert data into a MySQL database using Python, you can use the `INSERT INTO` statement with the `execute()` method provided by the MySQL Connector/Python module.

```
import mysql.connector

# establish a connection to the MySQL server
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="yourdatabase"
)
```

```
# create a cursor object to execute SQL statements
mycursor = mydb.cursor()

# define the data to be inserted
data = ("John", "Doe", "john.doe@example.com")

# prepare the SQL query
sql = "INSERT INTO customers (first_name, last_name, email) VALUES (%s, %s, %s)"

# execute the query with the data
mycursor.execute(sql, data)

# commit the transaction
mydb.commit()

# print the number of inserted rows
print(mycursor.rowcount, "record inserted.")
```

In this example, we first establish a connection to the MySQL server and create a cursor object to execute SQL statements. We then define the data to be inserted and prepare the SQL query using placeholders (%s) for the data values. Finally, we execute the query with the data, commit the transaction, and print the number of inserted rows.

Read

To read data from a MySQL database in Python, you can use the MySQL Connector module. Here's an example of how to read data from a MySQL database:

```
import mysql.connector

# establish a connection to the database
db = mysql.connector.connect(
  host="localhost",
  user="username",
  password="password",
  database="mydatabase" )

# create a cursor object to execute SQL queries
cursor = db.cursor()
# define the SQL query to retrieve data
sql = "SELECT * FROM customers"

# execute the query using the cursor
cursor.execute(sql)

# fetch all rows from the query result
rows = cursor.fetchall()
```

```
# iterate over the rows and print each row
for row in rows:
  print(row)

# close the database connection
db.close()
```

In this example, we first establish a connection to the MySQL database using the `mysql.connector` module. We then create a cursor object to execute SQL queries. We define an SQL query to retrieve data from the "customers" table, and then execute the query using the cursor. We fetch all rows from the query result using the `fetchall()` method, and then iterate over the rows and print each row. Finally, we close the database connection using the `close()` method.

Update

To update data in a MySQL database using Python, you can use the `UPDATE` statement. Here's an example code snippet:

```
import mysql.connector

# Connect to the database
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase")
```

```
# Create a cursor object
mycursor = mydb.cursor()

# Prepare the SQL query to update data
sql = "UPDATE customers SET address = 'Highway 2' WHERE name = 'John'"

# Execute the query
mycursor.execute(sql)

# Commit the changes to the database
mydb.commit()

# Print the number of rows affected by the update query
print(mycursor.rowcount, "record(s) affected")
```

In this example, we first connect to the MySQL database using the `mysql.connector` module. Then we create a cursor object that allows us to execute SQL queries. We prepare an `UPDATE` statement to update the address of the customer named "John" to "Highway 2". We execute the query using the cursor `execute()` method and commit the changes to the database using the `commit()` method. Finally, we print the number of rows affected by the update query using the cursor's `rowcount` attribute.

Delete

Sure, here's an example of how to delete data from a MySQL table using the Python MySQL Connector module:

```
import mysql.connector
```

```
# Connect to the database
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase")

# Create a cursor object to execute SQL queries
mycursor = mydb.cursor()

# Define the delete query and data
sql = "DELETE FROM customers WHERE address = %s"
val = ("Highway 37", )

# Execute the delete query with data
mycursor.execute(sql, val)
# Commit the changes to the database
mydb.commit()
# Print the number of rows deleted
print(mycursor.rowcount, "record(s) deleted")
```

In this example, we connect to a database, create a cursor object, and define a delete query that deletes all customers with the address "Highway 37". We then execute the query with the data using the `execute()` method, commit

the changes to the database with `commit()`, and print the number of rows deleted using `mycursor.rowcount`.

7.5 Other Module

There are many Python modules that can be useful for beginners, depending on what they are trying to accomplish. Here are a few that are commonly used:

1. **math:** Provides access to mathematical functions and constants.
2. **random:** Allows you to generate random numbers and choose random elements from a sequence.
3. **datetime:** Provides classes for working with dates and times.
4. **os:** Provides a way to interact with the file system and operating system.
5. **time:** Provides functions for working with time, such as measuring elapsed time and sleeping.
6. **sys:** Provides access to some variables and functions related to the Python interpreter.
7. **csv:** Provides functionality for working with CSV (comma-separated values) files.
8. **json:** Provides functionality for working with JSON (JavaScript Object Notation) data.
9. **requests:** Allows you to send HTTP requests and work with web APIs.

These are just a few examples, and there are many more Python modules available depending on your needs and interests.

Chapter 8

Problem definitions

Assignment: Coffee Shop Inventory Management System

You have been hired as a developer to create a program to help a small coffee shop manage their inventory and sales. The coffee shop sells three types of coffee beans: Arabica, Robusta, and Liberica. Your program should allow the owner to add new inventory, update existing inventory, and view sales data.

Requirements:

1. The program should have a menu-driven interface that allows the user to choose from the following options:
    a. Add new inventory
    b. Update existing inventory
    c. View sales data
    d. Exit
2. The program should store the inventory data in a dictionary where the keys are the type of coffee bean (Arabica, Robusta, Liberica) and the values are the quantity of each type of coffee bean in stock.
3. When the user chooses the "Add new inventory" option, the program should prompt the user to enter the type of coffee bean and the quantity to add to the inventory. If the type of coffee bean is not already in the inventory, it should be added with the specified quantity. If the type of coffee bean is already in the inventory, the quantity should be updated to reflect the additional inventory.
4. When the user chooses the "Update existing inventory" option, the program should display the current inventory and prompt the user to enter the type of coffee bean to update and the new quantity. If the type of coffee bean is not in the inventory, the program should display an error message.

5. When the user chooses the "View sales data" option, the program should display the sales data for each type of coffee bean. The sales data should be stored in a dictionary where the keys are the type of coffee bean and the values are the total sales for that type of coffee bean.

6. The program should allow the user to exit the program at any time by choosing the "Exit" option.

7. The program should handle any errors gracefully and display appropriate error messages to the user.

8. The program should be written in Python.

Submission:

1. Submit a Python script file containing your code.
2. Include a document that explains how to use the program and any assumptions you made in implementing the program.
3. Include a brief reflection on your experience developing the program, including any challenges you faced and how you overcame them.

Solution

```
class CoffeeShop:
    def __init__(self):
        self.inventory = {
            'Arabica': 0,
            'Robusta': 0,
            'Liberica': 0
        }
        self.sales = {
            'Arabica': 0,
            'Robusta': 0,
            'Liberica': 0
        }

    def add_inventory(self, coffee_type, amount):
        if coffee_type in self.inventory:
            self.inventory[coffee_type] += amount
            print(f"{amount} kg of {coffee_type} added to inventory.")
        else:
            print(f"{coffee_type} is not a valid coffee type.")

    def update_inventory(self, coffee_type, amount):
        if coffee_type in self.inventory:
            if self.inventory[coffee_type] < amount:
```

```
            print("Not enough inventory.")
        else:
            self.inventory[coffee_type] -= amount
            print(f"{amount} kg of {coffee_type} sold.")
            self.sales[coffee_type] += amount
    else:
        print(f"{coffee_type} is not a valid coffee type.")

  def view_inventory(self):
    print(f"Current inventory:")
    for coffee_type, amount in self.inventory.items():
        print(f"{coffee_type}: {amount} kg")

  def view_sales(self):
    print(f"Sales data:")
    for coffee_type, amount in self.sales.items():
        print(f"{coffee_type}: {amount} kg")

coffee_shop = CoffeeShop()

# add inventory
coffee_shop.add_inventory('Arabica', 10)
coffee_shop.add_inventory('Robusta', 5)
coffee_shop.add_inventory('Liberica', 3)
```

```
# update inventory and record sales
coffee_shop.update_inventory('Arabica', 5)
coffee_shop.update_inventory('Robusta', 3)

# view current inventory and sales data
coffee_shop.view_inventory()
coffee_shop.view_sales()
```

This program defines a `CoffeeShop` class with methods to add new inventory, update existing inventory, and view sales data. The `inventory` attribute is a dictionary that stores the current amount of each coffee type in the inventory, while the `sales` attribute is a dictionary that stores the amount of each coffee type sold.

The `add_inventory` method takes a coffee type and amount as arguments, and adds the specified amount to the inventory if the coffee type is valid. The `update_inventory` method takes a coffee type and amount as arguments, and updates the inventory and sales data if the coffee type is valid and there is enough inventory. The `view_inventory` method simply prints out the current inventory, and the `view_sales` method prints out the sales data.

In this example implementation, we've added 10 kg of Arabica, 5 kg of Robusta, and 3 kg of Liberica to the inventory using the `add_inventory` method. We then update the inventory and record sales by selling 5 kg of Arabica and 3 kg of Robusta using the `update_inventory` method. Finally, we view the current inventory and sales data using the `view_inventory` and `view_sales` methods.

This is just a basic example, and there are many other features that could be added to make the program more useful for managing a coffee shop's inventory and sales.

Chapter 1: Python Introduction

1. Write a Python program to print "Hello, World!" on the screen.

2. Write a Python program to ask the user for their name and then print a greeting message using their name.

3. Write a Python program to ask the user to enter two numbers and then print the sum of those two numbers.

4. Write a Python program to ask the user for their age and then calculate and print the year they were born in.

Chapter 2 Fundamentals of Python

1. Write a Python program to take two integers as inputs from the user and print their sum.

2. Write a Python program to take a string as input from the user and print it in reverse order.

3. Write a Python program to convert a temperature in Celsius to Fahrenheit. The formula for conversion is: F = (C * 9/5) + 32, where F is the temperature in Fahrenheit and C is the temperature in Celsius.

4. Write a Python program to take a list of numbers as input from the user and print the sum and average of the numbers.

Chapter 3 - List, Tuple, Dictionary, Set, Frozenset

1. Write a Python program to create a list of numbers from 1 to 50 and print all the even numbers from the list.

2. Write a Python program to find the intersection of two given lists and store the result in a new list.

3. Write a Python program to create a dictionary of names and ages of people. Then, print all the keys and values of the dictionary separately.

4. Write a Python program to remove the duplicates from a given list and store the result in a new list.

Chapter 4 (Loops and Control Statements)

1. Write a program that prompts the user to enter a number and prints all the even numbers from 0 to that number.

2. Write a program that prompts the user to enter a password and keeps prompting until the correct password is entered. The correct password is "password123".

3. Write a program that prompts the user to enter a list of numbers, separated by commas. The program should then print the sum of the numbers and the average of the numbers.

Chapter 5: Functions, Classes, and Exception Handling

1. Write a Python function to check if a given number is prime or not. The function should take a number as input and return a boolean value.

2. Create a class called "Rectangle" with attributes length and width. The class should have methods to calculate the area and perimeter of the rectangle.

3. Write a program that reads two numbers from the user and performs the four basic arithmetic operations (addition, subtraction, multiplication, and division) on them using separate functions for each operation.

4. Write a Python program to handle an exception when a user inputs a string instead of an integer in the input() function. The program should ask the user to enter the input again until a valid integer is provided.

Chapter 6 on File I/O, Regex, and Recursion

1. Write a Python program to read a file and print the number of words, lines, and characters in the file.

2. Write a Python program to extract all email addresses from a text file using regular expressions.

3. Write a Python program to find the factorial of a number using recursion.

4. Write a Python program to encrypt and decrypt a file using the Caesar Cipher.

5. Write a Python program to remove all the blank lines from a text file.

6. Write a Python program to count the occurrences of a word in a text file.

7. Write a Python program to convert a CSV file to an Excel file.

8. Write a Python program to check whether a given string is a palindrome or not using recursion.

9. Write a Python program to find the largest and smallest number in a text file.

10. Write a Python program to search for a pattern in a text file using regular expressions.

Chapter 7 Modules

1. Write a Python program to get the size, creation date, and absolute path of a file using the OS module.

2. Write a Python program to connect to a MySQL database using the PyMySQL module, create a new table, insert data into the table, and retrieve data from the table.

3. Write a Python program to get the current system's CPU utilization percentage and memory usage percentage using the psutil module.