Course Pack For

Python Programming

Course: BCA
Course Code: 501
Semester: V
Year: 2024-25

Course Leader: Mr. Nripesh Kumar Nrip
Course Instructor: Dr. Mansi Agnihotri
Mr. Nripesh Kumar Nrip

Mr. Nripesh Kumar Nrip	Dr. Daljeet Singh Bawa	Dr. Yamini Agarwal
Program Coordinator	Forwarded by: HOD	Approved By: Director

Bharati Vidyapeeth (Deemed to be University)
Institute of Management & Research, New Delhi
An ISO 9001: 2008 & 14001:2004 Certified Institute
A-4, Paschim Vihar, New Delhi-110063
(Ph: 011-25284396, 25285808, Fax No. 011-25286442)

BVIMR SNAPSHOT

Established in 1992, Bharati Vidyapeeth (Deemed to be University) Institute of Management and Research (BVIMR), New Delhi focuses on imbibing the said values across various stakeholders through adequate creation, inclusion, and dissemination of knowledge in management education.
The institute has over the past few years emerged in the lead with a vision of Leadership in professional education through innovation and excellence. This excellence is sustained by consistent value enhancement and initiation of value-added academic processes in institute's academic systems.

Based on the fabulous architecture and layout on the lines of Nalanda Vishwa Vidyalaya, the institute is a scenic marvel of lush green landscape with modern interiors. The Institute which is ISO 9001:2015 certified is under the ambit of Bharati Vidyapeeth University (BVU), Pune as approved by Govt. of India on the recommendation of UGC under Section 3 of UGC Act vide its letter notification No. F. 9 – 16 / 2004 – U3 dated 25th February 2005.

Strategically located in West Delhi on the main Rohtak Road, BVIMR, New Delhi has splendid layout on sprawling four acres of plot with 'state-of-art' facilities with all classrooms, Library Labs, Auditorium etc. that are fully air-conditioned. The Institute that has an adjacent Metro station "Paschim Vihar (East)", connects the entire Delhi and NCR.

We nurture our learners to be job providers rather than job seekers. This is resorted to by fostering the skill and enhancement of knowledge base of our students through various extracurricular, co-curricular and curricular activities by our faculty, who keep themselves abreast by various research and FDPs and attending Seminars/Conferences. The Alumni has a key role here by inception of SAARTHI Mentorship program who update and create professional environment for learners' centric academic ambiance and bridging industry-academia gap.

Our faculty make distinctive contribution not only to students but to Academia through publications, seminars, conferences apart from quality education. We also believe in enhancing corporate level interaction including industrial projects, undertaken by our students under continuous guidance of our faculty. These form the core of our efforts which has resulted in being one of the premier institutes of management.

At BVIMR, we are imparting quality education in management at Doctorate, Postgraduate and Undergraduate levels.

Mr. Nripesh Kumar Nrip (Assistant Professor, BVIMR)

Mr. Nripesh Kumar Nrip is a highly motivated academician and researcher with expertise in the field of Application of Computations. He is Working as an Assistant Professor in the Department of Computer Applications at Bharati Vidyapeeth (Deemed to be University) Pune and has 14 years of Academic Experience.

Currently, He is Working as an Assistant Professor in the Department of Computer Applications at Bharati Vidyapeeth Institute of Management and Research New Delhi.

He holds a Master Degree in Computer Applications (MCA) from Bharati Vidyapeeth Pune and has several years of research experience. He is also pursuing Ph.D. from Bharati Vidyapeeth Pune. His research interests lie in the areas of Multidisciplinary domains, especially the Applications of ICT in various domains like Agriculture, Education, Health, Rural development etc.

Prof. Nrip has authored/co-authored more than 27 research articles in peer-reviewed journals and conference proceedings, which have received over 35K reads and downloads. He has also presented his research work at several national and international conferences. He has authors 2 books. He has also published one patent and secured one Design Grant. Prof. Nrip is a reviewer for several renowned scientific journals in the field of his area of research interest. He is also an active member of International Association of Engineers (IAENG).

Dr. Mansi Agnihotri

Mobile M-91-9953871988
E-Mail magnihotri93@gmail.com

Experience and Accomplishments

Dr. Mansi Agnihotri earned her Ph.D. in March 2024 from the University School of Information, Communication and Technology (USICT), Guru Gobind Singh Indraprastha University (GGSIPU). Her thesis, titled "A New Metric to Identify Critically Affected Classes and Improving Their Quality Through Refactoring," presents a novel approach to software quality. She worked as a full-time research scholar at USICT, GGSIPU. She qualified UGC NET-JRF (DEC 2018) and availed UGC JRF and SRF during her Ph.D. She holds a Master's degree in Computer Applications.

She has a good experience of research and has published 6 research papers with SCIE, ESCI, Scopus indexing and has presented her work at 2 international conferences. She has 4 years of experience of teaching as a research scholar. She has been a part of organizing team for different Faculty Development Programme (FDP's) conducted at USIC&T, GGSIPU.

Index

Programme: BCA CBCS Revised Syllabus w.e.f.-Year 2022–2023			
Semester	**Course Code**	**Course Title**	
V	501	Python Programming	
	Prepared by	Dr.M.K.Patil	
Type	**Credits**	**Evaluation**	**Marks**
DSC	3	UE: IE	60:40

Course Objectives

- A Python programming course is designed to equip students with a comprehensive understanding of the language and its application.
- Starting with an introduction to Python's history and community, the course guides students through setting up their development environment and mastering fundamental syntax and data types.
- Students learn control flow structures, functions, and modules, progressing to file handling, object-oriented programming (OOP) principles, and data structures.
- The curriculum includes essential skills such as error handling, debugging, and the use of popular libraries and frameworks.
- Emphasis is placed on best practices, code style, collaborative development using version control (e.g., Git), testing, and debugging techniques.
- Overall, the objectives aim to empower students with a well-rounded skill set for effective Python programming and application development.

Course Outcome

CO1: Using some motivating examples to remember and quickly builds up basic concepts such as conditionals, loops, functions, lists, strings and tuples.

CO2:Students will get acquainted built in data structures in python, understand features and programming constructs of python language. During this course, they will understand main control structures of procedural programming languages.

CO3: They will make of function to reduce problem into small modules, To familiarize with exceptions and mechanism to handle it, make use of python to read and write data into files

CO4:Analyzing the different problems based on CSV files

CO5: Ability to choose appropriate data dictionary for problem solving

CO6: Design and create their own programs for solving a real life problem

1. The Detailed Syllabus

Unit	Contents	Sessions (Hrs.)	COs Number	Teaching Methodology	Cognition Level	Evaluation Tools
Introduction to Python:	History of Python, Unique features of Python, Python Identifiers, Keywords and	6	CO1, CO2, CO3	Classroom Teaching, ICT-based teaching	Remembering, Understanding, Applying	Assignments, Quizzes

	Indentation, Comments and document interlude in Python, Getting User Input Python, Data Types, variables, Python Core objects and Functions Number and Maths					
Statements and Control Structures:	Assignment statement, import statement, print statement, if: elif: else: statement, for: statement., while: statement., continue and break statements, try: except statement., raise statement., with statement, del, case statement	6	CO1, CO2, CO3	Classroom Teaching, ICT-based teaching	Remembering, Understanding, Applying	Lab Assignments
List, Ranges & Tuples & Dictionaries in Python	Introduction, Lists in Python, Understanding Iterators, Generators, Comprehensions and Lambda Expressions, Generators and Yield Next and Ranges, Understanding and using Ranges, Ordered Sets with tuples, Introduction to Python	4	CO1, CO2, CO3	Classroom Teaching, ICT-based teaching	Remembering, Understanding, Applying	Lab Assignments

	Dictionaries, Python Sets					
Functions, Modules, Packages, and Debugging Functions:	The def statement Returning values, Parameters, Arguments, Local variables, Other things to know about functions, Global variables and the global statement, Doc strings for functions, Decorators for functions, lambda Iterators and generators, Modules, Doc strings for modules, Packages	6	CO1, CO2, CO3, CO5	Classroom Teaching, ICT-based teaching	Remembering, Understanding, Applying, Evaluating	Lab Assignments
Python Object Oriented	Overview of OOP, Creating Classes and Objects, Accessing attributes Built-In Class Attributes, Destroying Objects	4	CO1, CO2	Classroom Teaching, ICT-based teaching	Remembering, Understanding	Lab Assignments
Python Exceptions Handling	What is Exception? Handling an exception try….except…else Try-finally clause Argumento fan Exception. Python Standard	6	CO1, CO2, CO3, CO5	Classroom Teaching, ICT-based teaching	Remembering, Understanding, Applying, Evaluating	Lab Assignments

	Exceptions Raising and exceptions, User-Defined Exceptions					
Input and Output in Python & Built in Functions	File Objects, creating a file object, reading File contents, writing data into file, reading and writing CSV files, using with clause, Using Exception handling with file operations,	6	CO1, CO2, CO3, CO5, CO6	Project-based teaching, ICT-based teaching	Remembering, Understanding, Applying, Analyzing, Evaluating	Lab Assignments, Live case study from the website Kaggle.com

2. CO-PO Mapping

CO/PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	-	-	1	-	2	-	-	-	-	-
CO2	2	3	2	-	1	-	2	-	-	-	-	-
CO3	3	3	3	3	2	1	2	-	-	-	-	-
CO4	1	3	3	-	2	-	-	-	-	-	-	-
CO5	1	3	3	3	3	1	-	-	-	-	-	-
CO6	-	-	-	1	1	1	-	-	-	-	-	1
CO.	**1.67**	**2.5**	**1.83**	**1.16**	**1.67**	**0.5**	**1**	**-**	**-**	**-**	**-**	**0.16**
CO	**2**	**3**	**2**	**1**	**2**	**1**	**1**					**0**

1- Low , 2- Medium, 3- High, If no correlation put '-'

3. Evaluation

Internals: 40%
Externals: 60%
Total : 100%

4. Assessment Mapping

Parameter	Marks	CO1	CO2	CO3	CO4	CO5	CO6
Class Participation/ Attendance	20	4	5	4	4	2	1
Assignments/ Projects	10	1	2	3	3	1	-
Test	10	2	2	2	3	1	-
Internal	40	7	9	9	10	4	1
End Term (Univ)	60						

5. Rationale for Mapping Program Outcomes and Course Outcomes:

CO1 & PO1 Mapped at 3	These objectives and outcomes provide students with a comprehensive understanding of programming fundamentals and their practical applications.
CO1 & PO 2 Mapped at 3	These objectives and outcomes prepare students to excel in both programming fundamentals and problem-solving skills in the field of computer science.
CO1 & PO5 Mapped at 1	These objectives and outcomes work together to prepare students to excel in both the foundational aspects of programming and the application of cutting-edge technology in software development.
CO1 & PO7 Mapped at 2	These objectives and outcomes work together to prepare students to excel in programming fundamentals and to maintain relevance and competitiveness as computing professionals.
CO2 & PO1 Mapped at 2	These focus on the importance of applying mathematical and computational knowledge to conceptualize and address problems in various domains, ensuring that students can apply what they've learned effectively.
CO2 & PO2 Mapped at 3	These objectives and outcomes prepare students for success in computer science and problem-solving.
CO2 & PO3 Mapped at 2	These objectives and outcomes prepare students for success in the field of computer science, problem-solving, and technology integration.
CO2 & PO5 Mapped at 1	These objectives and outcomes improve students' understanding and retention of core computer science concepts, dynamic programming, and more.

CO2 & PO7 Mapped at 2	These objectives and outcomes improve students' understanding and emphasize the need for ongoing learning and adaptation within the ever-evolving computing industry, preparing students to excel as computing professionals.
CO3 & PO1 Mapped at 3	These objectives and outcomes prepare students for proficiency in data structures and computational problem-solving.
CO3 & PO2 Mapped at 3	These objectives and outcomes prepare students to excel in data structures and problem-solving in the field of computer science.
CO3 & PO3 Mapped at 3	These objectives and outcomes prepare students for success in data structures, problem-solving, and technology integration in practical contexts.
CO3 & PO4 Mapped at 3	These objectives and outcomes provide students with a strong foundation in data structures and emphasizes the ability to use scientific methods to experiment, collect data, and draw meaningful conclusions, ensuring a comprehensive skill set in the field of computer science.
CO3 & PO5 Mapped at 2	These objectives and outcomes equip students with a strong foundation in software development in today's rapidly evolving technological landscape.
CO3 & PO6 Mapped at 1	These objectives and outcomes ensure that students are not only technically proficient but also ethically responsible in their computing practices.
CO3 & PO7 Mapped at 2	These objectives and outcomes ensure that students are well-prepared for success as computing professionals.
CO4 & PO1 Mapped at 1	These objectives and outcomes prepare students for proficiency in handling data in this common format.
CO4 & PO2 Mapped at 3	These objectives and outcomes prepare students for proficiency in working with CSV data and addressing related challenges.
CO4 & PO3 Mapped at 3	These objectives and outcomes prepare students to excel in data analysis and problem-solving in the context of emerging technologies and business scenarios.
CO4 & PO5 Mapped at 2	These objectives and outcomes focus on teaching students how to analyze problems based on CSV files, and techniques for developing innovative software solutions, ensuring students are well-prepared for success in data analysis.
CO5 & PO1 Mapped at 1	These objectives and outcomes ensure that students can make well-informed decisions about data structures in their problem-solving processes.
CO5 & PO2 Mapped at 3	These objectives and outcomes prepare students for proficiency in data structure selection and problem-solving in the field of computer science.
CO5 & PO3 Mapped at 3	These objectives and outcomes prepare students to excel in data structure selection and problem-solving using emerging technologies in practical contexts.
CO5 & PO4 Mapped at 3	These objectives and outcomes ensure students are well-prepared for effective data-driven problem-solving.
CO5 & PO 5 Mapped at 3	These objectives and outcomes prepare students for effective data-driven problem-solving and software development in a rapidly evolving technological landscape.
CO5 & PO 6	These objectives and outcomes prepare students for effective data-

Mapped at 1	driven problem-solving ensuring that students make responsible and compliant decisions in their computing practices.
CO6 & PO 4 Mapped at 1	These objectives and outcomes prepare students for innovative data structure development and problem-solving in various contexts.
CO6 & PO 5 Mapped at 1	These objectives and outcomes prepare students for creative and effective problem-solving in the field of computer science.
CO6 & PO 6 Mapped at 1	These objectives and outcomes prepare students for creative problem-solving within ethical and legal boundaries.
CO6 & PO 12 Mapped at 1	These objectives and outcomes prepare students to be innovative problem solvers and entrepreneurs.

6. Session plan

Session	Topic	Pedagogy	Learning Outcome
Module I: Introduction to Python:			
1	History of Python, Unique features of Python,	Lecture with PPT and Discussion	CO1, CO2, CO3
2	Python Identifiers, Keywords and Indentation,	Lecture with PPT and Discussion	CO1, CO2, CO3
3	Comments and document interlude in Python	Lecture with PPT and Discussion	CO1, CO2, CO3
4	Getting User Input Python, Data Types	Lecture ,Demonstration and Practical Exercise	CO1, CO2, CO3
5	Python Core objects	Handouts	CO1, CO2, CO3
6	Functions, Number and Maths	Demonstration Practical Exercise	CO1, CO2, CO3
Module II: Statements and Control Structures			
7	Assignment statement, import statement, print statement,	Lecture with PPT and Discussion	CO1, CO2, CO3
8	if: elif: else: statement,	Lecture with PPT and Discussion	CO1, CO2, CO3
9	if: elif: else: statement,	Lecture with PPT and Discussion	CO1, CO2, CO3
10	for: statement.,	Lecture ,Demonstration and Practical Exercise	CO1, CO2, CO3
11	while: statement., continue and break statements,	Demonstration Practical Exercise	CO1, CO2, CO3

12	try: except statement., raise statement., with statement, del, case statement	Demonstration Practical Exercise	CO1, CO2, CO3
Module III: List, Ranges & Tuples & Dictionaries in Python			
13	Introduction, Lists in Python,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3
14	Understanding Iterators, Generators,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3
15	Comprehensions and Lambda Expressions, Generators and Yield Next and Ranges,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3
16	Understanding and using Ranges, Understanding Ordered Sets with tuples,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3
17	Introduction to Python Dictionaries, Python Sets	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3
18	CES 1	Handouts	`
Module IV: **Functions, Modules, Packages, and Debugging Functions**			
19	The def statement Returning values, Parameters, Arguments.	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
20	Local variables, Other things to know about functions,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
21	Global variables and the global statement, Doc strings for functions,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
22	Decorators for functions,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
23	lambda Iterators and generators,	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
24	Modules, Doc strings for modules,	Lecture, Demonstration and Practical	CO1, CO2, CO3, CO5

		Exercise	
25	Packages	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
Module V: Python Object Oriented			
26	Overview of OOP	Lecture with PPT	CO1, CO2
27	Creating Classes and Objects,	Lecture, Demonstration and Practical Exercise	CO1, CO2
28	Accessing attributes	Lecture, Demonstration and Practical Exercise	CO1, CO2
29	Built-In Class Attributes,	Demonstration Practical Exercise	CO1, CO2
30	Destroying Objects	Demonstration Practical Exercise	CO1, CO2
Module VI: Python Exceptions Handling			
31	What is Exception? Handling an exception	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
32	except…else Try-finally clause	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
33	Argumento fan Exception.	Demonstration Practical Exercise	CO1, CO2, CO3, CO5
34	Raising and exceptions	Demonstration Practical Exercise	CO1, CO2, CO3, CO5
35	User-Defined Exceptions	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
36	Python Standard Exceptions	Lecture, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5
37	CES 2		
Module VI: Input and Output in Python & Built in Functions			
38	File Objects	Lecture with PPT,	CO1, CO2, CO3, CO5, CO6

		Demonstration and Practical Exercise	
39	creating a file object, reading File contents,	Demonstration and Practical Exercise	CO1, CO2, CO3, CO5, CO6
40	writing data into file, reading and writing CSV files,	Lecture with PPT, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5, CO6
41	writing data into file, reading and writing CSV files,	Lecture with PPT, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5, CO6
42	using with clause, Using Exception handling	Lecture with PPT, Demonstration and Practical Exercise	CO1, CO2, CO3, CO5, CO6
43	Using Exception handling with file operations	Lecture with PPT, Demonstration	CO1, CO2, CO3, CO5, CO6
44	Revision		
45	Revision		

7. Textbook:

1. Artificial Intelligence by Elaine Rich and Kevin Knight, Tata McGraw Hill
2. Understanding Machine Learning. Shai Shalev-Shwartz and Shai Ben-David. Cambridge University Press.
3. Artificial Neural Network, B. Yegnanarayana, PHI, 2005
Tom Mitchell, "Machine Learning", McGraw Hill, 1997
2. E. Alpaydin, "Introduction to Machine Learning", PHI, 2005.

8. Reference Book:

1. Christopher M. Bishop. Pattern Recognition and Machine Learning (Springer)
2. Introduction to Artificial Intelligence and Expert Systems by Dan W. Patterson, Prentice Hall of India
3. Andrew Ng, Machine learning yearning, https://www.deeplearning.ai/machine-learning-yearning/
4. Aurolien Geron," Hands-On Machine Learning with Scikit-Learn and TensorFlow, Shroff/O'Reilly",2017
5. Andreas Muller and Sarah Guido," Introduction to Machine Learning with Python: A Guide for Data Scientists", Shroff/O'Reilly, 2016

9. MOOC

a) **Swayam : https://onlinecourses.swayam2.ac.in/cec22_cs20/preview**
b) **NEPTEl https://onlinecourses.nptel.ac.in/noc19_cs41/preview**
c) **EDX : https://www.edx.org/learn/python**

Unit 1:
Introduction to Python

History of Python, Unique features of Python, Python Identifiers, Keywords and Indentation, Comments and document interlude in Python, Getting User Input Python, Data Types, variables, Python Core objects and Functions, Number and Maths.

History of Python

Python is a high-level, interpreted programming language created by Guido van Rossum. The language's development began in the late 1980s, with its first official release in February 1991. Python was designed to be easy to read and simple to implement, with an emphasis on code readability and simplicity.

Creation and Early Development

Guido van Rossum started working on Python during the Christmas holidays in 1989, as a hobby project to keep himself occupied. At the time, he was working at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. Python was intended as a successor to the ABC language, which was also developed at CWI and taught programming to non-programmers. Van Rossum aimed to fix some issues he found with ABC while retaining its strengths, such as ease of learning and readability.

Influences and Design Philosophy

Python's design was influenced by several programming languages:

- **ABC Language**: Python inherited ABC's structure and simplicity.
- **Modula-3**: Influenced Python's exception-handling features.
- **C and C++**: Provided syntax familiar to many programmers.
- **Algol-68**: Contributed to Python's design of procedural programming.
- **Smalltalk**: Influenced Python's object-oriented programming features.
- **Unix Shell**: Inspired Python's scripting capabilities.
- **Other Scripting Languages**: Python adopted features that made scripting and automating tasks more accessible.

Python's design philosophy emphasizes readability and simplicity. This philosophy is encapsulated in "The Zen of Python," a collection of aphorisms written by Tim Peters that capture the essence of Python's design. Some key principles include:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Readability counts.

Major Versions and Evolution

Python has undergone several major revisions, with the most significant being the transition from Python 2 to Python 3.

- **Python 1.0**: Released in January 1994, Python 1.0 included many features found in modern Python, such as exception handling, functions, and the core data types (str, list, dict, etc.).
- **Python 2.0**: Released in October 2000, Python 2.0 introduced list comprehensions, garbage collection, and the zip() function. Python 2 continued to evolve with many enhancements and became the dominant version for many years.
- **Python 3.0**: Released in December 2008, Python 3.0 was designed to rectify fundamental design flaws in the language. It was not backward compatible with Python 2. This version introduced several improvements, such as a more consistent and cleaned-up language, new syntax features like print() as a function, and improved Unicode support. The transition to Python 3 was gradual, with the community supporting both versions until Python 2 was officially discontinued on January 1, 2020.

Community and Ecosystem

One of Python's greatest strengths is its extensive and active community. The Python Software Foundation (PSF), a non-profit organization, manages Python's development and promotes its growth. The community contributes to Python's vast ecosystem of libraries and frameworks, which cater to diverse domains such as web development (Django, Flask), data science (NumPy, pandas, SciPy), machine learning (TensorFlow, scikit-learn), and more.

Python has grown from a hobby project to one of the most popular programming languages in the world, known for its ease of learning, readability, and versatility. Its ongoing development and the active involvement of the community ensure that Python continues to evolve, meeting the needs of modern software development and innovation. With its robust standard library, powerful frameworks, and extensive community support, Python remains a top choice for beginners and experienced developers alike.

Unique Features of Python

Readable and Maintainable Code

Python's readable and maintainable code is one of its most significant advantages. The language uses indentation to define code blocks instead of braces, which enhances readability and makes the code visually appealing and easy to understand. This focus on readability helps developers maintain and debug code more efficiently.

Dynamically Typed

Python is dynamically typed, meaning there is no need to declare variable types explicitly. This flexibility allows developers to write more concise code without worrying about the data types, thus speeding up the development process.

Interpreted Language

Python is an interpreted language, which means it executes code line by line. This feature simplifies debugging and testing since errors are detected and reported immediately, allowing for quicker iterations during development.

Large Standard Library

Another standout feature is Python's large standard library, which includes extensive modules and packages for various tasks such as web development, data analysis, machine learning, and more. This comprehensive library enables developers to perform complex tasks without writing extensive code from scratch, thereby enhancing productivity.

Portability

Python's portability is also noteworthy. It can run on various operating systems, including Windows, macOS, Linux, and more, without requiring any modifications to the code. This cross-platform compatibility makes Python a versatile choice for developers working in diverse environments.

Support for Multiple Paradigms

Python supports multiple programming paradigms, including object-oriented, procedural, and functional programming. This flexibility allows developers to choose the best paradigm for their specific use case, making Python adaptable to a wide range of programming tasks.

Extensive Community Support

Finally, Python benefits from extensive community support. It has a large, active community that contributes to its growth and provides ample resources, such as documentation, tutorials, and forums, to help developers. This strong community support ensures that developers can find help and solutions to their problems, facilitating continuous learning and improvement.

Python Identifiers, Keywords, and Indentation

Identifiers

In Python, identifiers are names used to identify variables, functions, classes, modules, and other objects. These names must adhere to certain rules to be valid. An identifier can start with a letter (A-Z, a-z) or an underscore (_), and it can be followed by letters, digits (0-9), or underscores. It is important to note that Python identifiers are **case-sensitive**, meaning that variable, Variable, and VARIABLE would be considered three distinct identifiers. This case sensitivity allows for more precise naming conventions but requires careful attention to avoid errors due to mismatched case usage.

Example:

```
# Valid identifiers
variable = 10
_variable = "underscore"
var123 = 3.14
# Invalid identifiers (will raise syntax errors)
# 1variable = 10
# var@123 = 3.1
```

Keywords

Keywords in Python are reserved words that have special meanings and cannot be used as identifiers. These keywords are integral to the language's syntax and are used to construct statements and define the structure and flow of the code. Examples of Python keywords *include if, else, while, for, def, and class* etc.. Each keyword serves a specific function; for instance, if and else are used for conditional statements, while and for are used for loops, and def and class are used to define functions and classes, respectively. Using keywords correctly is essential for writing syntactically correct and functional Python code.

Example:

```
# Using keywords in Python

# if-else conditional statement
if variable > 5:
    print("Variable is greater than 5")
else:
    print("Variable is 5 or less")
```

```
# while loop
count = 0
while count < 5:
    print(count)
    count += 1

# for loop
for i in range(5):
    print(i)

# Function definition
def my_function():
    print("Hello, World!")

# Class definition
class MyClass:
    def __init__(self, value):
        self.value = value
```

Indentation

Indentation in Python is a distinctive feature that sets it apart from many other programming languages. Unlike languages that use braces {} to define the scope of loops, functions, classes, and other code blocks, Python uses indentation to determine the grouping of statements. Each block of code within a structure must be consistently indented with the same number of spaces or tabs. This consistent indentation is crucial, as improper indentation can lead to syntax errors or logical errors that are difficult to debug. Indentation not only enhances code readability but also enforces a clean and uniform coding style, making Python code easier to read and maintain.

Example:

```
# Proper indentation in a function
def my_function():
    print("This is properly indented")
    if True:
        print("This is inside an if statement")

# Properly indented function
def my_other_function():
    print("This will not cause an error")  # Properly indented

# Indentation in a loop
for i in range(5):
    print(i)  # Properly indented inside the loop

# Call the functions to see the output
my_function()
my_other_function()
```

Comments and Documentation in Python

Single-line Comments

Single-line comments in Python are initiated with the hash character (#). Any text following the # on that line is considered a comment and is ignored by the Python interpreter. Single-line comments are often used to provide brief explanations or annotations within the code, making it easier to understand and maintain.

Example:

```
# This is a single-line comment
x = 5  # This is an inline comment explaining that x is assigned the value 5
```

Multi-line Comments

Multi-line comments in Python can be created using triple quotes, either """ or '''. Although these are technically multi-line strings, they can be used as comments if they are not assigned to any variable or used as docstrings. Multi-line comments are useful for commenting out large sections of code or for providing detailed explanations that span multiple lines.

Example:

```
"""
This is a multi-line comment.
It spans multiple lines.
"""
y = 10

'''
This is another multi-line comment.
It also spans multiple lines.
'''
z = 15
```

Docstrings

Docstrings are a special type of comment used for documenting modules, classes, and functions. They are typically enclosed within triple quotes (""" ... """ or ''' ... '''). Docstrings are not just ignored by the interpreter; they can be accessed programmatically via the __doc__ attribute, making them an integral part of Python's built-in documentation system. They provide a convenient way to describe the purpose and usage of a module, class, or function.

Example:

```
def add(a, b):
    """
    This function takes two numbers and returns their sum.

    Parameters:
    a (int, float): The first number.
    b (int, float): The second number.

    Returns:
    int, float: The sum of the two numbers.
    """
    return a + b

# Accessing the docstring
print(add.__doc__)
```

In the example above, the add function has a docstring that explains its purpose, parameters, and return value. The docstring provides clear documentation that can be accessed using the __doc__ attribute.

Best Practices for Comments and Documentation

- **Single-line comments** should be used sparingly to annotate specific lines or sections of code that may not be immediately clear.
- **Multi-line comments** are useful for commenting out blocks of code during testing or for providing extensive explanations.
- **Docstrings** should be used to document all public modules, classes, and functions. They should provide a clear and concise description of the entity's purpose, parameters, and return values.

Example

```
# Demonstrating comments and documentation in Python
"""
This script provides examples of single-line comments, multi-line comments, and docstrings.
"""
def add(a, b):
    """
    Returns the sum of a and b.
    Parameters:
    a (int, float): First number.
    b (int, float): Second number.
        Returns:
    int, float: Sum of a and b.
    """
    return a + b
def subtract(a, b):
    """
    Returns the difference of a and b.
     Parameters:
    a (int, float): First number.
    b (int, float): Second number.
        Returns:
    int, float: Difference of a and b.
    """
    return a - b
# Main part of the script
if __name__ == "__main__":
    x, y = 10, 5  # Initialize variables
    # Perform operations and print results
    print(f"The sum of {x} and {y} is {add(x, y)}")  # Addition
    print(f"The difference between {x} and {y} is {subtract(x, y)}")  # Subtraction
'''
End of the script: Demonstrated single-line comments, multi-line comments, and docstrings.
'''
```

Note: By following these practices of using proper comments, you can make your code more understandable and maintainable, which is especially important when working on larger projects or collaborating with others.

Getting User Input in Python

In Python, the input() function is used to take input from the user. The function reads a line from the input (usually from the user via the keyboard), converts it into a string, and returns it.

```
name = input("Enter your name: ")
print("Hello, " + name)
```

In this example:

- The input() function displays the prompt message "Enter your name: ".
- The user types in their name and presses Enter.
- The entered name is stored in the variable name.
- The program then prints "Hello, " followed by the entered name.

Data Types in Python

Python supports various data types, which can be categorized into several groups:

1. Numeric Types:

- **int**: Integer numbers, e.g., 1, 100, -42.
- **float**: Floating-point numbers, e.g., 1.0, 3.14, -0.001.
- **complex**: Complex numbers, e.g., 1+2j, -3+4j.

2. Sequence Types:

- **str**: Strings, a sequence of characters, e.g., "hello", 'Python'.
- **list**: A mutable sequence of items, e.g., [1, 2, 3], ['a', 'b', 'c'].
- **tuple**: An immutable sequence of items, e.g., (1, 2, 3), ('a', 'b', 'c').

3. Mapping Type:

- **dict**: A collection of key-value pairs, e.g., {'name': 'Alice', 'age': 25}.

4. Set Types:

- **set**: An unordered collection of unique items, e.g., {1, 2, 3}, {'a', 'b', 'c'}.
- **frozenset**: An immutable version of a set.

5. Boolean Type:

- **bool**: Represents True or False.

6. Binary Types:

- **bytes**: Immutable sequences of bytes.
- **bytearray**: Mutable sequences of bytes.
- **memoryview**: Memory views of byte data.

Variables in Python

Variables are used to store data that can be referenced and manipulated later in the program. Python variables are dynamic, meaning their type can change based on the value assigned.

Variable Assignment:

You can assign a value to a variable using the = operator.

```
x = 5
y = "Hello"
z = 3.14
```

Dynamic Typing:
Python allows variables to change type dynamically.

```
a = 10       # a is an integer
a = "Hello"  # a is now a string
```

Multiple Assignment:
You can assign multiple variables in one line.

```
a, b, c = 1, 2, 3
```

Example: Python code demonstrating data types and variables.

```
# Integer
num = 10
print(type(num))  # Output: <class 'int'>

# Float
pi = 3.14
print(type(pi))  # Output: <class 'float'>

# String
greeting = "Hello, World!"
print(type(greeting))  # Output: <class 'str'>

# List
numbers = [1, 2, 3, 4, 5]
print(type(numbers))  # Output: <class 'list'>

# Tuple
coordinates = (10.0, 20.0)
print(type(coordinates))  # Output: <class 'tuple'>

# Dictionary
person = {"name": "Alice", "age": 25}
print(type(person))  # Output: <class 'dict'>

# Set
unique_numbers = {1, 2, 3, 3, 4}
print(type(unique_numbers))  # Output: <class 'set'>
```

```
# Boolean
is_valid = True
print(type(is_valid))  # Output: <class 'bool'}
```

Example : Python code demonstrating bytes, bytesarray and memoryview

```
# Bytes Example
print("Bytes Example:")
# Creating a bytes object
b = bytes([65, 66, 67, 68])
print(b)  # Output: b'ABCD'

# Accessing elements in a bytes object
print(b[0])  # Output: 65
print(b[1])  # Output: 66

# Iterating through a bytes object
for byte in b:
    print(byte, end=' ')  # Output: 65 66 67 68
print("\n")

# Bytearray Example
print("Bytearray Example:")
# Creating a bytearray object
ba = bytearray([65, 66, 67, 68])
print(ba)  # Output: bytearray(b'ABCD')

# Modifying elements in a bytearray
ba[0] = 97  # Changing 'A' (65) to 'a' (97)
print(ba)  # Output: bytearray(b'aBCD')

# Adding elements to a bytearray
ba.append(69)  # Adding 'E' (69)
print(ba)  # Output: bytearray(b'aBCDE')

# Converting bytearray to bytes
b_converted = bytes(ba)
print(b_converted)  # Output: b'aBCDE'
print("\n")

# Memoryview Example
print("Memoryview Example:")
# Creating a bytes object
b_mv = bytes([65, 66, 67, 68, 69])

# Creating a memoryview object
mv = memoryview(b_mv)
print(mv)  # Output: <memory at 0x00000123456789AB>

# Accessing elements through memoryview
```

```
print(mv[0])  # Output: 65

# Slicing memoryview
mv_slice = mv[1:4]
print(mv_slice.tobytes())  # Output: b'BCD'

# Creating a bytearray and a memoryview
ba_mv = bytearray([65, 66, 67, 68, 69])
mv_ba = memoryview(ba_mv)

# Modifying the original bytearray through memoryview
mv_ba[0] = 97  # Changing 'A' (65) to 'a' (97)
print(ba_mv)  # Output: bytearray(b'aBCDE')
```

Python Core Objects and Functions

Core Objects

In Python, everything is an object, including data types like integers and strings, and even functions and classes. Python's core objects are built into the language and provide a wide range of functionalities. For example **int**: for Integer, **float**: for Floating-point numbers, **str**: for String type, **list**: for Mutable etc.

Python Core Functions

Python's core functions provide a powerful toolkit for various operations, ranging from basic input/output to complex data manipulation. Here's a detailed overview of some of the essential core functions in Python:

- **input()**: Prompts the user to enter a list of numbers separated by spaces.
- **map()** and **int()**: Converts the input string into a list of integers.
- **sum()**: Calculates the sum of the numbers in the list.
- **len()**: Calculates the length of the list.
- **sorted()**: Sorts the list of numbers.
- **print()**: Displays the original list, the sum of the numbers, the number of elements, and the sorted list.

```
# Program to take a list of numbers from the user, calculate the sum, and sort the list

# Step 1: Get user input as a string and convert it to a list of integers
user_input = input("Enter a list of numbers separated by spaces: ")

# Convert the input string to a list of strings
input_list = user_input.split()

# Convert the list of strings to a list of integers
numbers = list(map(int, input_list))

# Step 2: Calculate the sum of the list of numbers
total = sum(numbers)

# Step 3: Find the length of the list
length = len(numbers)
```

```
# Step 4: Sort the list of numbers
sorted_numbers = sorted(numbers)

# Output the results
print(f"Original list of numbers: {numbers}")
print(f"Sum of numbers: {total}")
print(f"Number of elements: {length}")
print(f"Sorted list of numbers: {sorted_numbers}")
```

****

Unit 2:
Statements and Control Structures

Assignment statement, import statement, print statement, if: elif: else: statement, for: statement., while: statement., continue and break statements, try: except statement., raise statement., with statement, del, case statement

2.1 Assignment Statement

In Python, an assignment statement is used to assign a value to a variable. The basic syntax of an assignment statement is:

variable_name = value

Here's a breakdown of the components:

1. **variable_name**: This is the name of the variable to which you want to assign a value. Variable names should be descriptive and follow the naming rules (e.g., start with a letter or underscore, can contain letters, digits, and underscores, and are case-sensitive).
2. **= (equals sign)**: This is the assignment operator. It assigns the value on its right to the variable on its left.
3. **value**: This is the value that you want to assign to the variable. It can be a number, string, list, or any other data type.

Key Points of Assignment Statements

1. **Creating Variables:**
   When you assign a value to a variable for the first time, Python automatically creates the variable.
   x = 10 (Creates a variable x and assigns the value 10 to it)

2. **Updating Variables:**
   You can update the value of an existing variable using an assignment statement.
   Example:
   x = 20 (Updates the value of x to 20)

3. **Dynamic Typing:**
   Python variables are dynamically typed, meaning the same variable can hold values of different types at different times.
   Example:
   x = 10 (x is an integer)
   x = "Hello" (Now x is a string)

4. **Multiple Assignment:**
   Python allows you to assign values to multiple variables in a single statement.
   Example:
   a, b, c = 1, 2, 3 (a is 1, b is 2, c is 3)

5. **Chained Assignment:**
   You can assign a single value to multiple variables simultaneously.
   Example:
   x = y = z = 100 (x, y, and z all get the value 100)

6. **Swapping Variables:**
   Python provides a simple way to swap the values of two variables without using a temporary variable.
   Example:
   a, b = b, a (Swaps the values of a and b)

Working with Lists and Assignment
Assignments can also involve complex data structures like lists, where you can assign multiple values at once or update specific elements.

```
# Assigning a list to a variable
my_list = [1, 2, 3, 4, 5]
# Updating a specific element
my_list[2] = 10  # Now my_list is [1, 2, 10, 4, 5]
# Assigning multiple variables from a list
a, b, c = my_list[:3]  # a is 1, b is 2, c is 10
```

Assignment with Expressions
Assignments can also involve expressions. Python will evaluate the expression on the right side of the = operator and then assign the result to the variable on the left side.

```
a = 5
b = 10
c = a + b  # c is 15
d = a * b  # d is 50
```

In summary, assignment statements in Python are fundamental to storing and updating values in variables. They are flexible and support various operations that make them powerful tools for managing data in your programs.

2.2 Import Statement

The import statement in Python is used to bring in modules, which are collections of functions, classes, and variables, so that you can use them in your current program. This helps you organize your code into manageable pieces and reuse code across different programs. We can import any module, either internal or external into our code using the import statement.

1. Basic Import
To import an entire module, you use the import statement followed by the module name. This is useful when you want to access various functions and variables from a module.

Syntax: import module_name

Example:
import math
In this example, we import the math module, which provides mathematical functions and constants. After importing, you can use the functions and constants provided by the math module.

Usage:
result = math.sqrt(16) print(result) # This prints 4.0
Here, we use the sqrt function from the math module to compute the square root of 16. The result is 4.0, which is then printed.

2. Import with Alias
You can assign an alias to a module when importing it. This can make your code more concise, especially if the module name is long or if you use the module frequently.

Syntax: import module_name as alias

Example:
import numpy as np

In this example, we import the numpy module and give it the alias np. This allows us to reference numpy functions and variables using np.

Usage:
np_array = np.array([1, 2, 3]) print(np_array) # This prints [1 2 3]
Here, we create a numpy array using the array function from the numpy module, which we refer to as np due to the alias.

3. Import Specific Functions or Variables
If you only need certain functions or variables from a module, you can import them directly. This can make your code cleaner and more efficient by avoiding the need to reference the module name each time.

Syntax: from module_name import function_name

Example:
from math import sqrt, pi

In this example, we import the sqrt function and the pi constant from the math module. This allows us to use them directly without prefixing them with math.

Usage:
result = sqrt(16) print(result) # This prints 4.0
print(pi) # This prints 3.141592653589793
Here, we use the sqrt function to compute the square root of 16, and we print the value of pi.

4. Import All Functions and Variables
To import all functions and variables from a module, you can use the asterisk (*). This is generally discouraged due to potential name conflicts, as it imports everything into the current namespace.

Syntax: from module_name import *

Example:
from math import *

In this example, we import all functions and variables from the math module. This allows us to use them directly without prefixing them with math..

Usage:
result = sqrt(16) print(result) # This prints 4.0
pi_value = pi print(pi_value) # This prints 3.141592653589793
Here, we use the sqrt function to compute the square root of 16 and print the value of pi.

5. Importing Submodules
Modules can be organized into submodules, which can be imported individually. This is useful for working with specific parts of a larger module.
Syntax: import module_name.submodule_name

Example:
import os.path

In this example, we import the path submodule from the os module. The os module provides a way of using operating system dependent functionality like reading or writing to the file system, and path provides operations on pathnames.

Usage:
basename = os.path.basename('/usr/local/bin/python') print(basename) # This prints 'python'

Here, we use the basename function from the path submodule to extract the base name from the given file path.

6. Importing from a Package
Python packages are directories containing multiple modules. You can import specific modules from a package, which helps in organizing and managing code efficiently.
Syntax: from package_name import module_name

Example:
from collections import deque

In this example, we import the deque class from the collections module. The collections module implements specialized container datatypes providing alternatives to Python's general-purpose built-in containers like dict, list, set, and tuple.

Usage:
d = deque([1, 2, 3]) print(d) # This prints deque([1, 2, 3])

Here, we create a deque (double-ended queue) object using the deque class from the collections module.

2.3 Print Statement

The print() function in Python outputs data to the standard output device (usually the console). It is versatile and can handle various data types and formats. By default, it prints each argument provided to it and ends with a newline.

Parameters of the print() Statement

The print() function includes several parameters that control how data is output. Here's a detailed breakdown of each parameter:

1. **object**:
   Description: This represents the values or variables to be printed. You can pass one or more objects separated by commas. These objects are converted to strings and displayed in the output.
   Example: print("Hello", 123, [1, 2, 3]) prints Hello 123 [1, 2, 3].
2. **sep**:
   Description: This parameter specifies the string that separates multiple objects. By default, it is a single space (' '), but you can change it to any string.
   Example: print("Python", "is", "fun", sep="-") prints Python-is-fun.
3. **end**:
   Description: This parameter determines what is printed at the end of the output. The default is a newline character ('\n'), which moves the cursor to the next line. You can set it to any string to change this behavior.
   Example: print("Hello", end=", ") followed by print("world!") prints Hello, world! on the same line.
4. **file**:
   Description: This parameter specifies a file-like object where the output should be directed. By default, it is sys.stdout, which sends output to the console. You can redirect output to other file objects.
   Example: with open('output.txt', 'w') as file: print("Writing to a file", file=file) writes Writing to a file to the file output.txt.
5. **flush**:
   Description: This boolean parameter, when set to True, forces the stream to be flushed immediately, ensuring that the output appears without buffering delays. By default, it is False.
   Example: print("Immediate flush", flush=True) ensures that the output is flushed and visible immediately.

Key Points

- **Conversion**: The print() function automatically converts non-string objects to strings using their __str__() method.
- **Formatting**: The sep and end parameters provide flexibility in formatting output, allowing for customized separation and termination of printed lines.
- **Output Redirection**: The file parameter enables output redirection to files or other file-like objects.
- **Immediate Output**: The flush parameter ensures that output is immediately visible, useful for real-time logging.

Formatted strings (f-strings)

- Formatted strings in Python, often referred to as f-strings, are a way to embed expressions inside string literals, using curly braces {}.
- They provide a concise and readable way to include the value of variables and expressions within strings.
- F-strings were introduced in Python 3.6 and have since become a popular method for string formatting due to their simplicity and efficiency.

Basic Usage

To create an f-string, prefix the string with the letter f or F. Inside the string, use curly braces {} to include the variables or expressions you want to embed.

SYNTAX: f"string {expression}"

Example

```
name = "Alice"
age = 30
print(f"Name: {name}, Age: {age}")
```

Explanation:

- f"Name: {name}, Age: {age}" is an f-string where {name} and {age} are replaced by their values directly.

Key Points

- **Flexibility**: Each method offers different levels of flexibility and readability. % formatting is less common now, str.format() provides more control, and f-strings are the most modern and convenient.
- **Readability**: F-strings are often preferred for their readability and performance, especially when dealing with multiple variables or complex expressions.
- **Expressions**: F-strings can evaluate expressions inside the curly braces, allowing for dynamic content within strings

2.4 If: elif: else statement

In Python, the if and elif statements are used for conditional execution of code. They allow a program to make decisions and execute different blocks of code based on specific conditions. This is fundamental for controlling the flow of a program.

The if Statement

The if statement is used to test a condition and execute a block of code if that condition is true. If the condition is false, the code block is skipped.

Syntax:

```
if condition:
    # Block of code to execute if the condition is true
```

- **condition**: An expression that evaluates to either True or False. This can be a comparison, a boolean expression, or any expression that returns a boolean value.

Example:

```
age = 18
if age >= 18:
    print("You are an adult.")
```

Explanation:

- The condition age >= 18 is evaluated. Since age is 18, the condition is true.
- The code inside the if block (print("You are an adult.")) is executed.

The elif Statement

The elif statement stands for "else if." It is used when there are multiple conditions to check. The elif block is only executed if its condition is true and all preceding if and elif conditions were false.

Syntax:

```
if condition1:
    # Block of code if condition1 is true
elif condition2:
    # Block of code if condition2 is true and condition1 was false
```

- **condition1**: The first condition to test. If it evaluates to True, the corresponding block of code is executed.
- **condition2**: The second condition to test if condition1 is false. Additional elif blocks can be added to test more conditions.

Example:

```
temperature = 30
if temperature > 30:
    print("It's very hot.")
elif temperature > 20:
    print("It's warm.")
```

Explanation:

- The first condition temperature > 30 is tested. Since temperature is 30, this condition is false.
- The elif condition temperature > 20 is then tested. Since temperature is 30, this condition is true.
- The code inside the elif block (print("It's warm.")) is executed.

Using else with if and elif

The else statement can be used to define a block of code that is executed when none of the preceding if or elif conditions are true.

Syntax:

```
if condition1:
```

```
    # Block of code if condition1 is true
elif condition2:
    # Block of code if condition2 is true and condition1 was false
else:
    # Block of code if none of the above conditions are true
```

Example:

```
number = 15
if number > 20:
    print("Number is greater than 20.")
elif number > 10:
    print("Number is greater than 10 but less than or equal to 20.")
else:
    print("Number is 10 or less.")
```

Explanation:

- The if condition number > 20 is false.
- The elif condition number > 10 is true, so print("Number is greater than 10 but less than or equal to 20.") is executed.
- If number had been 10 or less, the else block would have executed.

Combining Multiple Conditions

You can combine multiple conditions using logical operators such as and, or, and not.

Example:

```
age = 25
has_license = True
if age >= 18 and has_license:
    print("You can drive.")
else:
    print("You cannot drive.")
```

Explanation:

- The condition age >= 18 and has_license checks if both age is at least 18 and has_license is True.
- If both conditions are met, print("You can drive.") is executed. Otherwise, print("You cannot drive.") is executed.

Key Points

- **Conditional Testing**: The if statement is used to test conditions and execute code based on those conditions.
- **elif for Multiple Conditions**: The elif statement allows for multiple conditions to be checked in sequence.
- **Default Case with else**: The else statement provides a default block of code to execute when none of the preceding conditions are true.
- **Logical Operators**: Use logical operators to combine multiple conditions for more complex decision-making.

2.5 For Statement

The for statement in Python is used for iterating over a sequence (such as a list, tuple, dictionary, set, or string) or other iterable objects. It is a versatile control flow statement that allows you to execute a block of code repeatedly for each item in the sequence.

Syntax
The basic syntax of the for loop is:
for variable in iterable:
    # Block of code to execute for each item in the iterable

- **variable**: A variable that takes the value of each item in the iterable one by one.
- **iterable**: An object that can return its members one at a time. Common examples include lists, tuples, strings, and ranges.

Examples

1. **Iterating Over a List**

Code:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Explanation:

- The for loop iterates over the fruits list.
- fruit takes each value from the list ("apple", "banana", "cherry") in each iteration.
- The print(fruit) statement prints each fruit.

2. **Iterating Over a Range of Numbers**

Code:

```
for number in range(5):
    print(number)
```

Explanation:

- range(5) generates a sequence of numbers from 0 to 4.
- The for loop iterates over this range, and number takes each value from 0 to 4.

3. **Iterating Over a String**

Code:

```
for char in "hello":
    print(char)
```

Explanation:

- The for loop iterates over each character in the string "hello".
- char takes each character ("h", "e", "l", "l", "o") in each iteration.

4. **Iterating Over a Dictionary**

Code:

```
ages = {"Alice": 30, "Bob": 25, "Charlie": 35}
```

```
for name, age in ages.items():
    print(f"{name} is {age} years old")
```

Explanation:

- ages.items() returns a view of dictionary items as (key, value) pairs.
- The for loop iterates over these pairs, with name taking the dictionary keys and age taking the corresponding values.

5. **Using else with a for Loop**

Code:

```
for i in range(3):
    print(i)
else:
    print("Loop finished")
```

Explanation:

- The else block is executed after the for loop has iterated over all items, provided that the loop was not terminated by a break statement.

Advanced Usage

1. **Nested for Loops**

Code:

```
for i in range(2):
    for j in range(3):
        print(f"i={i}, j={j}")
```

Explanation:

- The outer for loop iterates over range(2), and for each iteration of the outer loop, the inner for loop iterates over range(3).
- This results in a total of 6 iterations, with i taking values 0 and 1, and j taking values 0, 1, and 2.

2. **List Comprehensions**

Code:

```
squares = [x**2 for x in range(5)]
print(squares)
```

Explanation:

- This list comprehension creates a list of squares of numbers from 0 to 4.
- The result is [0, 1, 4, 9, 16].

3. **Enumerate for Index and Value**

Code:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

Explanation:

- enumerate(fruits) provides both the index and the value during iteration.
- index and fruit take the index and value of each item, respectively.

Key Points

- **Iteration**: The for loop simplifies iterating over sequences and other iterable objects.
- **Flexible Iteration**: You can iterate over lists, strings, dictionaries, ranges, and more.
- **Nested Loops**: You can nest for loops to handle multi-dimensional data or complex iterations.
- **Comprehensions**: List comprehensions provide a concise way to create lists based on existing lists or ranges.

2.6 While Statement

The while statement in Python is used to create a loop that executes a block of code as long as a specified condition remains true. It is a fundamental control structure for repeating actions in Python.

Syntax

The basic syntax of a while loop is:

```
while condition:
    # Block of code to execute while condition is true
```

- **condition**: An expression that is evaluated before each iteration of the loop. The loop continues to execute as long as this condition evaluates to True. When the condition evaluates to False, the loop terminates.

Detailed Explanation

1. **Condition Evaluation**
    - Before each iteration, the condition is checked.
    - If the condition is True, the block of code inside the while loop is executed.
    - After executing the block of code, the condition is evaluated again.
    - This process repeats until the condition evaluates to False.

2. **Loop Execution**
    - The block of code inside the loop must be indented to denote that it is part of the loop.
    - It is important to ensure that the condition eventually becomes False, or the loop will continue indefinitely.

3. **Example**

Code:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Explanation:

Initialization: count is initialized to 0.
Condition Check: The condition count < 5 is evaluated.
First Iteration: count is 0, which is less than 5, so the print(count) statement executes, printing 0. count is then incremented by 1 (new value is 1).
Second Iteration: The condition count < 5 is checked again. Now count is 1, which is still less than 5. The print(count) statement prints 1, and count is incremented to 2.
This process continues until count reaches 5.
Termination: When count becomes 5, the condition count < 5 evaluates to False, and the loop exits. The block of code within the while loop is no longer executed.

4. **Execution Flow**
   - The while loop starts by evaluating the condition.
   - If the condition is True, it enters the loop, executes the code block, and re-evaluates the condition.
   - This process continues until the condition becomes False.
   - Once the condition is False, the loop terminates, and the program continues with the code following the loop (if any).

5. **Important Considerations**
   Condition Validity: Ensure that the condition will eventually become False to avoid creating an infinite loop.
   Code Block Execution: The code block should perform operations that will eventually lead to the condition becoming False.

Example with a String
Code:

```
index = 0
word = "Python"
while index < len(word):
    print(word[index])
    index += 1
```

Explanation:

- **Initialization**: index is set to 0, and word is "Python".
- **Condition Check**: The condition index < len(word) is evaluated. len(word) returns 6 (length of the string "Python").
  - **First Iteration**: index is 0, so word[index] is "P". This is printed, and index is incremented to 1.
  - **Subsequent Iterations**: The loop prints each character of "Python" until index is 6, at which point the condition index < len(word) becomes False.
- **Termination**: When index reaches 6, the loop terminates, and the program continues.

2.7 Continue and break statements.

The continue Statement in Python

The continue statement is used to skip the rest of the code inside a loop for the current iteration and proceed to the next iteration of the loop. It is commonly used to control the flow of loops when certain conditions are met, allowing you to bypass specific code.

Syntax
continue
Detailed Explanation

1. **Functionality**
    - **Skipping Code**: When the continue statement is executed, it immediately stops the execution of the remaining code in the current iteration of the loop.
    - **Next Iteration**: The loop then proceeds to the next iteration, re-evaluating the loop's condition.

2. **Common Use Cases**
    - **Filtering Data**: When you need to skip certain values or conditions and process only the desired ones.
    - **Avoiding Execution**: To skip over parts of the loop code based on specific criteria without breaking out of the loop.

3. **Behavior with for Loops**
    - **Iteration Control**: In a for loop, the continue statement will skip the rest of the code inside the loop for the current item and proceed to the next item in the sequence.

Example:

```
for num in range(10):
    if num % 2 == 0:  # Check if the number is even
        continue     # Skip the rest of the loop body for even numbers
    print(num)       # Print only odd numbers
```

Explanation:

- **Iteration**: The loop iterates over numbers from 0 to 9.
- **Condition**: num % 2 == 0 checks if the number is even.
- **Action**: If the number is even, continue skips the print(num) statement for that iteration.
- **Output**: 1, 3, 5, 7, 9 (only odd numbers are printed).

4. **Behavior with while Loops**
    - **Condition Re-evaluation**: In a while loop, continue will skip the rest of the loop body and go back to re-evaluate the loop condition.

Example:

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:  # Check if the count is even
        continue       # Skip the rest of the loop body for even counts
```

```
    print(count)       # Print only odd counts
```

Explanation:

- **Initialization**: count starts at 0 and is incremented by 1 in each iteration.
- **Condition**: count % 2 == 0 checks if the count is even.
- **Action**: If count is even, continue skips the print(count) statement.
- **Output**: 1, 3, 5, 7, 9 (only odd counts are printed).

Key Points

- **Execution Flow**: The continue statement affects only the current iteration of the loop, not the entire loop.
- **Loop Condition**: After continue, the loop condition is re-evaluated to determine if further iterations are needed.

The break Statement in Python

The break statement is used to exit from a loop prematurely. It immediately terminates the loop and transfers control to the first statement following the loop. It is useful for stopping the loop based on a specific condition or when a particular condition is met.

Syntax

break

Detailed Explanation

1. **Functionality**
   - **Immediate Termination**: When the break statement is executed, the loop is terminated immediately.
   - **Transfer Control**: Control is transferred to the statement immediately after the loop.

2. **Common Use Cases**
   - **Exiting Early**: When you need to stop the loop as soon as a certain condition is met, such as finding an item in a search loop.
   - **Breaking out of Nested Loops**: To break out of multiple nested loops, though this typically requires additional logic or the use of functions.

3. **Behavior with for Loops**
   - **Immediate Exit**: In a for loop, break exits the loop immediately, regardless of the loop's condition or the remaining items.

Example:

```
for num in range(10):
    if num == 5:   # Check if the number is 5
        break      # Exit the loop when num is 5
    print(num)     # Print numbers less than 5
```

Explanation:

- **Iteration**: The loop iterates over numbers from 0 to 9.
- **Condition**: if num == 5 checks if the current number is 5.

- **Action**: When num is 5, break exits the loop.
- **Output**: 0, 1, 2, 3, 4 (numbers less than 5 are printed; 5 and subsequent numbers are not printed).

4. **Behavior with while Loops**
    - **Immediate Exit**: In a while loop, break also exits the loop immediately, regardless of the condition.

Example:

```
count = 0
while count < 10:
    if count == 5:  # Check if count is 5
        break      # Exit the loop when count is 5
    print(count)   # Print counts less than 5
    count += 1
```

Explanation:

- **Initialization**: count starts at 0 and is incremented by 1 in each iteration.
- **Condition**: if count == 5 checks if the count is 5.
- **Action**: When count is 5, break exits the loop.
- **Output**: 0, 1, 2, 3, 4 (counts less than 5 are printed; 5 and subsequent counts are not printed).

Key Points

- **Immediate Effect**: break exits the loop immediately, regardless of any remaining iterations or conditions.
- **Loop Termination**: The loop terminates, and execution resumes with the statement following the loop.

2.8 Try:except statement

The try and except statements in Python are used for exception handling. Exception handling allows a program to deal with runtime errors in a controlled manner without crashing. By using try and except, you can handle errors gracefully and provide useful feedback to the user.

Syntax

The basic syntax of try and except is:

```
try:
    # Code that might raise an exception
    # This is where you write the code that you want to test for exceptions

except ExceptionType:
    # Code that runs if an exception of ExceptionType occurs
    # This is where you handle the exception
```

- try Block: Contains code that might cause an exception.

- except Block: Contains code that runs if an exception occurs. You can specify the type of exception to catch.

Detailed Explanation

1. Purpose
    - Error Handling: To manage and handle exceptions that occur during program execution.
    - Graceful Degradation: To ensure the program continues to run or provides meaningful feedback when an error occurs.
2. Handling Specific Exceptions
    - You can specify which type of exception to handle by replacing ExceptionType with the actual exception class.
    - This allows you to handle different types of exceptions differently.

Example:

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
```

Explanation:

- try Block: The code attempts to convert user input to an integer and divide 10 by that integer.
- except ZeroDivisionError: Handles the case where the user enters 0, causing a division by zero.
- except ValueError: Handles the case where the user input is not a valid number.

3. Handling Multiple Exceptions
    - You can have multiple except blocks to handle different exceptions.

Example:

```
try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter valid numbers.")
```

Explanation:

- try Block: The code takes two numbers as input and tries to divide them.
- except ZeroDivisionError: Handles division by zero.
- except ValueError: Handles invalid input.

4. Using else with try and except
    - The else block can be used to specify code that should run if no exceptions are raised in the try block.

Example:

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
else:
    print(f"The result is {result}.")
```

Explanation:

- else Block: Executes if no exceptions occur in the try block. Here, it prints the result if the division is successful.

5. Using finally with try and `except
    - The finally block contains code that always runs, regardless of whether an exception was raised or not.

Example:

```
try:
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("Error: The file was not found.")
finally:
    file.close()  # Ensures the file is closed whether or not an exception occurred
```

Explanation:

- finally Block: Executes regardless of whether an exception was raised. In this example, it ensures that the file is closed after attempting to read it.

6. Handling Multiple Exceptions in a Single except Block
    - You can handle multiple exceptions with a single except block by specifying a tuple of exceptions.

Example:

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except (ZeroDivisionError, ValueError) as e:
    print(f"Error: {e}")
```

Explanation:

- except (ZeroDivisionError, ValueError) as e: Handles both ZeroDivisionError and ValueError. The variable e holds the exception message.

Key Points

- Exception Handling: Helps to manage errors and prevent program crashes.
- Specific Exceptions: Handle different types of exceptions separately for more precise error handling.
- else Block: Executes code if no exceptions are raised in the try block.
- finally Block: Ensures that code runs regardless of whether an exception occurred, useful for cleanup actions.

Aspect	try Statement	except Statement
Purpose	To define a block of code that might raise an exception.	To handle exceptions that are raised in the try block.
Usage	Encapsulates code that is potentially error-prone.	Specifies how to respond if an error occurs in the try block.
Syntax	try: # Code that might raise an exception	except ExceptionType: # Code to handle the exception
Execution Flow	The code inside the try block is executed first.	The code inside the except block is executed if an exception occurs.
Error Handling	Identifies where exceptions might occur.	Provides the mechanism to handle specific exceptions.
Exception Catching	Does not catch exceptions itself; it only defines where they might occur.	Catches and processes exceptions raised in the try block.
Placement	Must be placed before the except block.	Must be placed after the try block.
Code Example	try: num = int(input("Enter a number: ")) result = 10 / num	except ZeroDivisionError: print("Cannot divide by zero.")
Optional Use	Optional to use in code if no potential errors need to be handled.	Must be used in conjunction with try to handle exceptions.
Common Combination	Used with except, else, and finally for complete exception handling.	Can be used with try, else, and finally for robust error management.

******

UNIT 3

List, Ranges & Tuples & Dictionaries in Python

Topic No.	Topic
3.1	Introduction
3.2	Lists in Python
3.3	Understanding Iterators
3.4	Generators
3.5	Comprehensions and Lambda Expressions
3.6	Generators and Yield Next and Ranges
3.7	Understanding and using Ranges
3.8	Ordered Sets with tuples
3.9	Introduction to Python Dictionaries
3.10	Python Sets
3.11	Del statement
3.12	Case statement

3.1 Introduction

In Python, data structures and functional tools play a crucial role in organizing, storing, and processing data efficiently. Understanding these concepts is fundamental to writing effective and optimized Python code. This introduction provides a concise overview of essential data structures and functional programming tools: lists, tuples, ranges, sets, dictionaries, iterators, generators, comprehensions, and lambda functions.

Lists

- **Definition**: A list is an ordered, mutable collection of elements. Lists can contain elements of different types, including other lists.
- **Key Features**: Dynamic sizing, supports indexing, slicing, and various methods for manipulation (e.g., append, remove, sort)

Tuples

- **Definition**: A tuple is an ordered, immutable collection of elements. Like lists, tuples can contain mixed data types.
- **Key Features**: Fixed size, useful for read-only collections, supports indexing and slicing.

Ranges

- **Definition**: A range represents an immutable sequence of numbers, commonly used for looping a specific number of times.
- **Key Features**: Memory efficient, typically used in for loops, generated using the range() function.

Sets

- **Definition**: A set is an unordered collection of unique elements. Sets are mutable and do not allow duplicate values.
- **Key Features**: Supports mathematical set operations like union, intersection, and difference, efficient for membership testing.

Dictionaries

- **Definition**: A dictionary is an unordered collection of key-value pairs. Keys must be unique and immutable, while values can be of any type.
- **Key Features**: Fast lookup, insertion, and deletion of elements based on keys, dynamic sizing.

Iterators

- **Definition**: An iterator is an object that represents a stream of data, allowing traversal through all the elements in a collection one at a time.
- **Key Features**: Implements the iterator protocol with the __iter__() and __next__() methods, used in loops and comprehensions.

Generators

- **Definition**: A generator is a type of iterator that generates values on the fly and uses the yield statement to return values one at a time.
- **Key Features**: Memory efficient, especially for large datasets, supports lazy evaluation.

Comprehensions

- **Definition**: Comprehensions provide a concise way to create collections like lists, sets, and dictionaries using a syntactic construct.
- **Key Features**: Readable and expressive, can include conditional logic and nested loops, improves code brevity.

Lambda Functions

- **Definition**: A lambda function is an anonymous, small function defined using the lambda keyword, typically used for short, throwaway functions.
- **Key Features**: Single-line functions, can have any number of arguments but only one expression, often used with functions like map(), filter(), and sorted().

Summary

These data structures and functional tools form the backbone of Python programming, enabling developers to write clean, efficient, and readable code. Lists, tuples, ranges, sets, and dictionaries provide various ways to store and manage collections of data, while iterators, generators, comprehensions, and lambda functions offer powerful tools for functional programming and data processing. Understanding these concepts is essential for mastering Python and leveraging its full potential.

Introduction to Strings

In Python, a string is a sequence of characters enclosed within single quotes ('), double quotes ("), or triple quotes (''' or """). Strings are immutable, meaning once a string is created, it cannot be modified.

Creating Strings

You can create strings using single, double, or triple quotes.

```
# Single and double quotes
single_quoted_string = 'Hello, World!'
double_quoted_string = "Hello, World!"
# Triple quotes for multi-line strings
triple_quoted_string = '''This is a
multi-line string.'''
```

Accessing Characters in Strings

Characters in a string can be accessed using indexing and slicing. Python uses zero-based indexing.

```
my_string = "Hello, World!"
# Accessing individual characters
first_char = my_string[0]    # 'H'
last_char = my_string[-1]    # '!'
sub_string = my_string[0:5]  # 'Hello'
```

String Immutability

Strings in Python are immutable, which means their content cannot be changed after creation. However, you can create new strings based on modifications.

```
original_string = "Hello"
new_string = original_string + " World"  # "Hello World"
```

Common String Operations

Concatenation

Strings can be concatenated using the + operator.

```
greeting = "Hello"
name = "Alice"
combined_string = greeting + ", " + name  # "Hello, Alice"
```

Repetition

Strings can be repeated using the * operator.

```
repeated_string = "Hi! " * 3  # "Hi! Hi! Hi! "
```

Membership Testing
You can test for the presence of a substring using the in keyword.
text = "Hello, World!"
result = "Hello" in text # True

String Methods
Python provides a rich set of methods to manipulate strings:

- str.lower(): Converts all characters to lowercase.
- str.upper(): Converts all characters to uppercase.
- str.title(): Converts the first character of each word to uppercase.
- str.capitalize(): Converts the first character to uppercase and the rest to lowercase.
- str.strip(): Removes leading and trailing whitespace.
- str.lstrip(): Removes leading whitespace.
- str.rstrip(): Removes trailing whitespace.
- str.split(separator): Splits the string into a list of substrings based on a separator.
- str.join(iterable): Joins elements of an iterable with the string as a separator.
- str.replace(old, new): Replaces occurrences of a substring with another substring.
- str.find(substring): Returns the lowest index of the substring if found, otherwise returns -1.
- str.count(substring): Returns the number of non-overlapping occurrences of the substring.

Examples of String Methods
Here are some examples demonstrating the usage of common string methods:

```
text = " Hello, World! "

# Changing case
lowercase_text = text.lower()      # " hello, world! "
uppercase_text = text.upper()      # " HELLO, WORLD! "
title_text = text.title()          # " Hello, World! "

# Trimming whitespace
stripped_text = text.strip()       # "Hello, World!"
left_stripped_text = text.lstrip() # "Hello, World! "
right_stripped_text = text.rstrip()# " Hello, World!"

# Splitting and joining
words = stripped_text.split(",")   # ['Hello', ' World!']
joined_text = "-".join(words)      # "Hello- World!"

# Replacing and finding
replaced_text = stripped_text.replace("World", "Python")  # "Hello, Python!"
index = stripped_text.find("World")  # 7
count = stripped_text.count("o")     # 2
```

Formatting Strings

Python provides several ways to format strings:

Using the format() Method
The format() method allows insertion of variables into strings.

```
name = "Alice"
age = 30
formatted_string = "Name: {}, Age: {}".format(name, age)  # "Name: Alice, Age: 30"
```

Using f-Strings (Python 3.6+)
f-Strings provide a concise way to embed expressions inside string literals.

```
name = "Alice"
age = 30
formatted_string = f"Name: {name}, Age: {age}"  # "Name: Alice, Age: 30"
```

Multi-line Strings
Triple quotes are used for multi-line strings, which can span multiple lines.

```
multi_line_string = """This is a
multi-line
string."""
```

Escape Sequences
Escape sequences are used to include special characters in strings.

- \\: Backslash
- \': Single quote
- \": Double quote
- \n: Newline
- \t: Tab

```
escaped_string = "She said, \"Hello!\"\nIt's a new line."
```

Example: Combining String Operations
Here's an example demonstrating various string operations:

```
# Initializing a string
original_text = " Python Programming "

# Trimming whitespace
trimmed_text = original_text.strip()

# Changing case
lowercase_text = trimmed_text.lower()
uppercase_text = trimmed_text.upper()

# Replacing substrings
replaced_text = trimmed_text.replace("Python", "Java")

# Splitting and joining
words = trimmed_text.split()
```

```
joined_text = "-".join(words)

# Formatting strings
name = "Alice"
language = "Python"
formatted_string = f"{name} loves {language}"

# Output the results
print("Original Text:", original_text)
print("Trimmed Text:", trimmed_text)
print("Lowercase Text:", lowercase_text)
print("Uppercase Text:", uppercase_text)
print("Replaced Text:", replaced_text)
print("Words:", words)
print("Joined Text:", joined_text)
print("Formatted String:", formatted_string)
```

f-Strings (Formatted String Literals) (String Interpolation)

Introduced in Python 3.6, f-strings provide a concise and convenient way to embed expressions inside string literals using curly braces {}. They are prefixed with an f or F.

Basic Usage

With f-strings, you can directly include variables and expressions inside a string.

```
name = "Alice"
age = 30
formatted_string = f"Name: {name}, Age: {age}"
print(formatted_string)  # Output: Name: Alice, Age: 30
```

Including Expressions

You can include any valid Python expression inside the curly braces.

```
result = f"5 + 3 = {5 + 3}"
print(result)  # Output: 5 + 3 = 8
```

Calling Functions

You can call functions inside f-strings.

```
def greet(name):
    return f"Hello, {name}!"
greeting = f"Greeting: {greet('Alice')}"
print(greeting)  # Output: Greeting: Hello, Alice!
```

Formatting Numbers

You can use format specifiers within f-strings to format numbers.

```
value = 1234.56789
formatted_value = f"Value: {value:.2f}"  # Output: Value: 1234.57
```

Multiline f-Strings

You can use f-strings in multi-line strings with triple quotes.

```
name = "Alice"
age = 30
formatted_string = f"""
Name: {name}
Age: {age}
"""
print(formatted_string)
# Output:
# Name: Alice
# Age: 30
```

The format() Method

The format() method of strings is a versatile way to create formatted strings, allowing for more control over the formatting compared to f-strings.

Basic Usage

You can use curly braces {} as placeholders and pass the values as arguments to the format() method.

```
name = "Alice"
age = 30
formatted_string = "Name: {}, Age: {}".format(name, age)
print(formatted_string)  # Output: Name: Alice, Age: 30
```

Positional and Keyword Arguments

You can specify the position of the arguments or use keywords.

```
# Positional arguments
formatted_string = "Name: {0}, Age: {1}".format(name, age)
print(formatted_string)  # Output: Name: Alice, Age: 30

# Keyword arguments
formatted_string = "Name: {name}, Age: {age}".format(name="Bob", age=25)
print(formatted_string)  # Output: Name: Bob, Age: 25
```

Reusing Arguments

You can reuse the same argument multiple times.

```
formatted_string = "Name: {0}, Age: {1}, Again Name: {0}".format(name, age)
print(formatted_string)  # Output: Name: Alice, Age: 30, Again Name: Alice
```

Formatting Numbers

You can format numbers using format specifiers.

```
value = 1234.56789
formatted_value = "Value: {:.2f}".format(value)
print(formatted_value)  # Output: Value: 1234.57
```

Aligning Text
You can align text within a specific width.

```
left_aligned = "{:<10}".format("left")
right_aligned = "{:>10}".format("right")
center_aligned = "{:^10}".format("center")
print(f"|{left_aligned}|{right_aligned}|{center_aligned}|")
# Output:
# |left      |     right|  center  |
```

Using Dictionaries
You can format strings using dictionaries.

```
data = {"name": "Alice", "age": 30}
formatted_string = "Name: {name}, Age: {age}".format(**data)
print(formatted_string)  # Output: Name: Alice, Age: 30
```

Comparison of f-Strings and format()

- **Readability**: f-Strings are generally more readable and concise than the format() method, especially for simple cases.
- **Performance**: f-Strings tend to be faster because they are evaluated at runtime and directly interpolated.
- **Flexibility**: The format() method provides more flexibility for complex formatting needs, such as reusing arguments and using dictionaries.

Raw Strings in Python
In Python, raw strings are used to treat backslashes (\) as literal characters rather than escape characters. This is particularly useful when dealing with strings that contain backslashes, such as file paths on Windows or regular expressions.

Syntax
A raw string is created by prefixing the string literal with an r or R.

```
raw_string = r"Your raw string here"
```

Example
Consider a typical string and a raw string to see the difference:

```
# Normal string with escape characters
normal_string = "This is a newline character: \n"

# Raw string
raw_string = r"This is a newline character: \n"

print("Normal String:", normal_string)
print("Raw String:", raw_string)
```

Output

```
Normal String: This is a newline character:

Raw String: This is a newline character: \n
```

Explanation

1. **Normal String**:
    - In the normal_string, the \n is interpreted as a newline character, so the output will actually break the line where \n appears.
2. **Raw String**:
    - In the raw_string, the r prefix tells Python to treat backslashes as literal characters. Therefore, \n is displayed as \n in the output rather than being interpreted as a newline.

SOME INPORTANT PROGRAMS

Write Python Code to Determine Whether the Given String Is a Palindrome or Not Using Slicing

```
1. def main():
2. user_string = input("Enter string: ")
3. if user_string == user_string[::-1]:
4. print(f"User entered string is palindrome")
5. else:
6. print(f"User entered string is not a palindrome")
7. if __name__ == "__main__":
8. main()
```

Output

Case 1:
Enter string: madam
User entered string is palindrome
Case 2:
Enter string: cat
User entered string is not a palindrome

Program to Demonstrate String Traversing Using the *for* Loop

```
1. def main():
2. alphabet = "google"
3. index = 0
4. print(f"In the string '{alphabet}'")
5. for each_character in alphabet:
6. print(f"Character '{each_character}' has an index value of {index}")
7. index += 1
8. if __name__ == "__main__":
9. main()
```

Output

In the string 'google'
Character 'g' has an index value of 0
Character 'o' has an index value of 1
Character 'o' has an index value of 2
Character 'g' has an index value of 3
Character 'l' has an index value of 4
Character 'e' has an index value of 5

Program to Print the Characters Which Are Common in Two Strings

```
1. def common_characters(string_1, string_2):
2. for letter in string_1:
3. if letter in string_2:
4. print(f"Character '{letter}' is found in both the strings")
5. def main():
6. common_characters('rose', 'goose')
7. if __name__ == "__main__":
8. main()
```

Output

```
Character 'o' is found in both the strings
Character 's' is found in both the strings
Character 'e' is found in both the strings
```

Write Python Program to Count the Total Number of Vowels, Consonants and Blanks in a String

```
1. def main():
2. user_string = input("Enter a string: ")
3. vowels = 0
4. consonants = 0
5. blanks = 0
6. for each_character in user_string:
7. if(each_character == 'a' or each_character == 'e' or each_character == 'i' or
each_character == 'o' or each_character == 'u'):
8. vowels += 1
9. elif "a" < each_character < "z":
10. consonants += 1
11. elif each_character == " ":
12. blanks += 1
13. print(f"Total number of Vowels in user entered string is {vowels}")
14. print(f"Total number of Consonants in user entered string is {consonants}")
15. print(f"Total number of Blanks in user entered string is {blanks}")
16. if __name__ == "__main__":
17. main()
```

Output

```
Enter a string: may god bless you
Total number of Vowels in user entered string is 5
Total number of Consonants in user entered string is 9
Total number of Blanks in user entered string is 3
```

Write Python Program to Calculate the Length of a String Without Using Built-In *len()* Function

```
1. def main():
2. user_string = input("Enter a string: ")
3. count_character = 0
4. for each_character in user_string:
5. count_character += 1
6. print(f"The length of user entered string is {count_character} ")
7. if __name__ == "__main__":
8. main()
```

Output

Enter a string: To answer before listening that is folly and shame
The length of user entered string is 50

Write Python Program to Count the Occurrence of User-Entered Words in a Sentence

```
1. def count_word(word_occurrence, user_string):
2. word_count = 0
3. for each_word in user_string.split():
4. if each_word == word_occurrence:
5. word_count += 1
6. print(f"The word '{word_occurrence}' has occurred {word_count} times")
7. def main():
8. input_string = input("Enter a string ")
9. user_word = input("Enter a word to count its occurrence ")
10. count_word(user_word, input_string)
11. if __name__ == "__main__":
12. main()
```

Output

Enter a string You cannot end a sentence with because because because is a conjunction
Enter a word to count its occurrence because
The word 'because' has occurred 3 times

3.2 Lists in Python

Overview:

- **Lists** are a fundamental data structure in Python. They are ordered, mutable collections of items, which means you can change their content after creation. Lists can contain items of different data types, including other lists.

Operations on Lists

1. **Creating Lists**:
    - **Empty List**:

```
empty_list = [
]
```

Explanation: This creates an empty list. You can later add items to this list using methods like append().

- **List with Initial Elements**:

```
numbers = [1, 2, 3, 4, 5]
mixed_list = [1, "text", 3.14, True, [1, 2, 3]]
```

Explanation: numbers is a list of integers, while mixed_list contains different data types, including another list.

2. **Accessing Elements**:
    - **Indexing**:

```
first_element = numbers[0]  # Outputs: 1
last_element = numbers[-1]  # Outputs: 5
```

Explanation: Lists are zero-indexed. Index 0 gives the first element, and -1 gives the last element. Python supports negative indexing to count from the end of the list.

    - **Slicing**:

```
sublist = numbers[1:4]  # Outputs: [2, 3, 4]
```

Explanation: Slicing extracts a subset of the list. The slice 1:4 starts at index 1 and goes up to, but does not include, index 4.

3. **Modifying Lists**:
    - **Appending**:

```
numbers.append(6)  # Adds 6 to the end of the list
```
Explanation: append() adds an element to the end of the list.

    - **Inserting**:

```
numbers.insert(2, 2.5)  # Inserts 2.5 at index 2
```
Explanation: insert(index, element) inserts an element at a specified position in the list, shifting subsequent elements.

    - **Removing**:

```
numbers.remove(2.5)  # Removes the first occurrence of 2.5
```
Explanation: remove() deletes the first occurrence of a specified value from the list.

    - **Popping**:

```
last = numbers.pop()  # Removes and returns the last element
```
Explanation: pop() removes the element at the specified index (or the last element if no index is provided) and returns it.

4. **Sorting and Reversing**:
    - **Sorting**:

numbers.sort() # Sorts the list in place
sorted_list = sorted(numbers) # Returns a new sorted list
Explanation: sort() arranges the list in ascending order. sorted() returns a new list and leaves the original unchanged.

- **Reversing**:

numbers.reverse() # Reverses the list in place
Explanation: reverse() changes the order of elements in the list to the reverse of their original order.

5. **List Comprehensions**:

squares = [x**2 for x in range(10)] # Outputs: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Explanation: List comprehensions provide a concise way to create lists. The expression [x**2 for x in range(10)] generates squares of numbers from 0 to 9.

- **Conditional List Comprehension**:

even_squares = [x**2 for x in range(10) if x % 2 == 0] # Outputs: [0, 4, 16, 36, 64]
Explanation: This example includes a condition (if x % 2 == 0) to filter elements. Only squares of even numbers are included in the resulting list.

3.3 Understanding Iterators

Overview:

- **Iterators** are objects that implement the iterator protocol, consisting of __iter__() and __next__() methods. They allow you to traverse through a container's elements without exposing its underlying structure.

Creating and Using Iterators

1. **Creating an Iterator**:
   - **Custom Iterator**:

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

rev = Reverse('giraffe')
for char in rev:
    print(char)  # Outputs: e, f, a, r, g
```

Explanation:

- **__init__()**: Initializes the iterator with data and an index.
- **__iter__()**: Returns the iterator object itself.
- **__next__()**: Returns the next item in the sequence. If the index is 0, it raises StopIteration to signal that there are no more items.

2. **Using Built-in Iterators**:
    - **Lists**:

```
my_list = [1, 2, 3]
it = iter(my_list)
print(next(it))  # Outputs: 1
print(next(it))  # Outputs: 2
```

Explanation:

- **iter()**: Converts a list into an iterator.
- **next()**: Retrieves the next item from the iterator.

3.4 Generators

Overview:

- **Generators** are a special type of iterator created using functions with the yield keyword. They allow you to iterate over a sequence of values, producing them one at a time and maintaining their state.

Creating and Using Generators

1. **Creating a Generator Function**:
    - **Simple Generator**:

```
def count_up_to(max):
   count = 1
   while count <= max:
      yield count
      count += 1

gen = count_up_to(5)
for number in gen:
   print(number)  # Outputs: 1, 2, 3, 4, 5
```

Explanation:

- **yield**: Pauses the function and returns a value to the caller, while maintaining the function's state.
- **count += 1**: Updates the state for the next value.

2. **Generator Expressions**:
    - **Using Generator Expressions**:

```
gen_expr = (x**2 for x in range(5))
print(list(gen_expr))  # Outputs: [0, 1, 4, 9, 16]
```

Explanation:

- **(x**2 for x in range(5))**: Creates a generator expression that computes squares of numbers from 0 to 4. Unlike list comprehensions, it doesn't store all values in memory.

3.5 Comprehensions and Lambda Expressions

Overview:

- **Comprehensions** provide a concise way to create lists, dictionaries, or sets. **Lambda expressions** are anonymous functions defined with the lambda keyword.

Comprehensions

1. **List Comprehensions**:
    - **Basic List Comprehension**:

```
squares = [x**2 for x in range(10)]  # Outputs: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Explanation: Constructs a list of squares from 0 to 9 in a single line. The expression x**2 generates the value, and for x in range(10) iterates through values.

    - **Conditional List Comprehension**:

```
even_squares = [x**2 for x in range(10) if x % 2 == 0]  # Outputs: [0, 4, 16, 36, 64]
```

Explanation: Adds a condition to include only squares of even numbers.

2. **Dictionary Comprehensions**:
    - **Basic Dictionary Comprehension**:

```
square_dict = {x: x**2 for x in range(5)}  # Outputs: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Explanation: Creates a dictionary where keys are numbers from 0 to 4 and values are their squares.

3. **Set Comprehensions**:
    - **Basic Set Comprehension**:

```
unique_squares = {x**2 for x in range(5)}  # Outputs: {0, 1, 4, 9, 16}
```

Explanation: Similar to list comprehensions but generates a set, which automatically removes duplicates.

Lambda Expressions

1. **Basic Lambda Expression**:
    - **Simple Lambda**:

```
add = lambda x, y: x + y
print(add(3, 5))  # Outputs: 8
```

Explanation: Defines an anonymous function that adds two numbers. Lambda functions are used for short, throwaway functions where a full function definition might be overkill.

2. **Lambda with Map**:
    - **Using Lambda with map()**:

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))  # Outputs: [1, 4, 9, 16]
```

Explanation: Applies a lambda function that squares each element in the numbers list using map().

3. **Lambda with Filter**:
    - **Using Lambda with filter()**:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))  # Outputs: [2, 4]
```

Explanation: Filters out elements that don't satisfy the lambda function's condition (in this case, even numbers).

3.6 Generators and Yield Next and Ranges

Overview:

- **Generators** use yield to produce a sequence of values over time. The yield statement allows a function to return a value and resume where it left off, maintaining its state.

Using Generators with yield

1. **Basic Generator Function**:
    - **Function with yield**:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)  # Outputs: 5, 4, 3, 2, 1
```

Explanation: countdown yields numbers from n down to 1. Each call to next() resumes from where it left off.

2. **Using yield with Iterators**:
    - **Combination of yield and __next__()**:

```
class Countdown:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        while self.n > 0:
            yield self.n
            self.n -= 1

for number in Countdown(5):
    print(number)  # Outputs: 5, 4, 3, 2, 1
```

Explanation: Combines yield with an iterator class. The __iter__() method uses yield to generate values.

Understanding Ranges

1. **Using range()**:
    - **Basic Usage**:

```
for i in range(5):
    print(i)  # Outputs: 0, 1, 2, 3, 4
```

Explanation: range() generates a sequence of numbers. By default, it starts at 0 and increments by 1.

- **Specifying Start, Stop, and Step**:

```
for i in range(2, 10, 2):
    print(i)  # Outputs: 2, 4, 6, 8
```

Explanation: range(start, stop, step) generates numbers starting from 2, ending before 10, with a step of 2.

3.7 Understanding and Using Ranges

Overview:

- **range()** is a built-in function that generates a sequence of numbers. It's often used in for loops to iterate over a sequence of numbers efficiently.

Range Parameters

1. **Single Argument**:
    - **Range from 0 to n**:

```
for i in range(5):
    print(i)  # Outputs: 0, 1, 2, 3, 4
```

Explanation: range(5) generates numbers from 0 to 4.

2. **Two Arguments**:

- **Range from start to stop**:

```
for i in range(2, 6):
   print(i)  # Outputs: 2, 3, 4, 5
```

Explanation: range(2, 6) starts at 2 and goes up to but does not include 6.

3. **Three Arguments**:
    - **Range with Step**:

```
for i in range(1, 10, 2):
   print(i)  # Outputs: 1, 3, 5, 7, 9
```

Explanation: range(1, 10, 2) starts at 1, stops before 10, and increments by 2.

3.8 Ordered Sets with Tuples

Overview:

- **Tuples** are immutable ordered collections of items. They are similar to lists but cannot be changed once created. **Sets** are unordered collections of unique elements.

Using Tuples

1. **Creating Tuples**:
    - **Simple Tuple**:

```
my_tuple = (1, 2, 3, 4)
```

Explanation: Creates a tuple with four integers. Tuples are created with parentheses.

- **Nested Tuples**:

```
nested_tuple = (1, (2, 3), 4)
```

Explanation: A tuple containing another tuple as an element.

2. **Accessing Elements**:

- **Indexing**:

```
element = my_tuple[1]  # Outputs: 2
```

Explanation: Accesses the element at index 1 of the tuple.

- **Slicing**:

```
sub_tuple = my_tuple[1:3]  # Outputs: (2, 3)
```

Explanation: Extracts elements from index 1 to 2.

3. **Tuple Methods**:
    - **Count**:

```
count = my_tuple.count(2)  # Outputs: 1
```

Explanation: Counts the occurrences of 2 in the tuple.

- **Index**:

```
index = my_tuple.index(3)  # Outputs: 2
```

Explanation: Finds the index of the first occurrence of 3.

Using Sets

1. **Creating Sets**:
    - **Simple Set**:

my_set = {1, 2, 3, 4}
Explanation: Creates a set with unique elements. Sets are created with curly braces.

    - **Empty Set**:

empty_set = set()
Explanation: Creates an empty set. Using curly braces alone {} creates an empty dictionary.

2. **Set Operations**:
    - **Adding Elements**:

my_set.add(5) # Adds 5 to the set
Explanation: Adds an element to the set. If the element is already present, no change occurs.

    - **Removing Elements**:

my_set.remove(3) # Removes 3 from the set
Explanation: Removes a specified element. Raises a KeyError if the element is not present.

    - **Set Operations**:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union = set1 | set2  # Outputs: {1, 2, 3, 4, 5}
intersection = set1 & set2  # Outputs: {3}
difference = set1 - set2  # Outputs: {1, 2}
```

Explanation: Performs union, intersection, and difference operations on sets.

3.9 Introduction to Python Dictionaries

Overview:

- **Dictionaries** are mutable, unordered collections of key-value pairs. Keys must be unique and immutable (e.g., strings, numbers, tuples).

Creating Dictionaries

1. **Basic Dictionary**:
    - **Creating**:

my_dict = {'name': 'Alice', 'age': 25}
Explanation: Creates a dictionary with keys 'name' and 'age' and their corresponding values.

2. **Accessing and Modifying Values**:
    - **Accessing Values**:

name = my_dict['name'] # Outputs: Alice
Explanation: Retrieves the value associated with the key 'name'.

    - **Modifying Values**:

my_dict['age'] = 26 # Updates the value for key 'age'
Explanation: Changes the value associated with the key 'age'.

- **Adding New Key-Value Pairs**:

my_dict['city'] = 'New York
4o mini

Program to Create and Modify a List:

```
my_list = [1, 2, 3, 4, 5]
my_list.append(6)
my_list.insert(2, 2.5)
my_list.remove(4)
print(my_list)  # Outputs: [1, 2, 2.5, 3, 5, 6]
```

Program to Slice and Access List Elements:

```
my_list = ['a', 'b', 'c', 'd', 'e']
first_element = my_list[0]
last_element = my_list[-1]
sub_list = my_list[1:4]
print("First element:", first_element)  # Outputs: a
print("Last element:", last_element)    # Outputs: e
print("Sliced list:", sub_list)         # Outputs: ['b', 'c', 'd']
```

Program to Use List Comprehensions:

```
squares = [x**2 for x in range(10)]
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print("Squares:", squares)         # Outputs: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print("Even squares:", even_squares)  # Outputs: [0, 4, 16, 36, 64]
```

Program to Create and Use a Custom Iterator:

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index -= 1
        return self.data[self.index]
```

```
rev = Reverse('giraffe')
for char in rev:
    print(char)  # Outputs: e, f, a, r, g
```

Program to Use Built-in Iterators:

```
my_list = [1, 2, 3, 4]
it = iter(my_list)
print(next(it))  # Outputs: 1
print(next(it))  # Outputs: 2
```

Program to Use Iterators in a Loop:

```
my_list = [1, 2, 3, 4]
it = iter(my_list)
for item in it:
    print(item)  # Outputs: 1, 2, 3, 4
```

Program to Create a Simple Generator:

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

for num in count_up_to(5):
    print(num)  # Outputs: 1, 2, 3, 4, 5
```

Program to Use a Generator with a Loop:

```
def fibonacci(n):
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a + b

for num in fibonacci(10):
    print(num)  # Outputs: 0, 1, 1, 2, 3, 5, 8
```

Program to Chain Generators:

```
def count_up_to(max):
    count = 1
```

```
    while count <= max:
        yield count
        count += 1

def squared_numbers(numbers):
    for number in numbers:
        yield number ** 2

for square in squared_numbers(count_up_to(5)):
    print(square)  # Outputs: 1, 4, 9, 16, 25
```

Program to Use List Comprehension with Condition:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)  # Outputs: [2, 4]
```

Program to Use Set Comprehension:

```
squares = {x**2 for x in range(5)}
print(squares)  # Outputs: {0, 1, 4, 9, 16}
```

Program to Use Lambda Expressions with map():

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Outputs: [1, 4, 9, 16]
```

Program to Use Generator with yield:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)  # Outputs: 5, 4, 3, 2, 1
```

Program to Use yield in a Custom Iterator:

```
class Countdown:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        while self.n > 0:
            yield self.n
            self.n -= 1
```

```
for number in Countdown(5):
    print(number)  # Outputs: 5, 4, 3, 2, 1
```

Program to Use range() with Step:

```
for i in range(1, 10, 2):
    print(i)  # Outputs: 1, 3, 5, 7, 9
```

Program to Use range() for Iteration:

```
for i in range(5):
    print(i)  # Outputs: 0, 1, 2, 3, 4
```

Program to Use range() with Start and Stop:

```
for i in range(2, 6):
    print(i)  # Outputs: 2, 3, 4, 5
```

Program to Use range() with Negative Step:

```
for i in range(10, 0, -2):
    print(i)  # Outputs: 10, 8, 6, 4, 2
```

Program to Create and Access Tuples:

```
my_tuple = (1, 2, 3, 4)
print(my_tuple[0])  # Outputs: 1
print(my_tuple[-1]) # Outputs: 4
```

Program to Create and Access Nested Tuples:

```
nested_tuple = (1, (2, 3), 4)
print(nested_tuple[1][0])  # Outputs: 2
```

Program to Perform Set Operations:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union = set1 | set2
intersection = set1 & set2
difference = set1 - set2
print("Union:", union)           # Outputs: {1, 2, 3, 4, 5}
print("Intersection:", intersection)  # Outputs: {3}
print("Difference:", difference)  # Outputs: {1, 2}
```

Program to Create and Modify a Dictionary:

```
my_dict = {'name': 'Alice', 'age': 25}
my_dict['age'] = 26
my_dict['city'] = 'New York'
print(my_dict)  # Outputs: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

Program to Access and Remove Dictionary Elements:

```
my_dict = {'name': 'Bob', 'age': 30, 'job': 'Engineer'}
name = my_dict.get('name')
my_dict.pop('job')
print(name)  # Outputs: Bob
print(my_dict)  # Outputs: {'name': 'Bob', 'age': 30}
```

Program to Iterate Through Dictionary Items:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
for key, value in my_dict.items():
    print(f"{key}: {value
```

UNIT 4
Functions, Modules, Packages, and Debugging Functions:

Topic No.	Topic
4.1	The def statement
4.2	Returning values
4.3	Parameters
4.4	Arguments
4.5	Local variables
4.6	Global variable and global statement
4.7	Doc strings for functions
4.8	Decorators for functions
4.9	Lambda Iterators and generators
4.10	Modules
4.11	Doc strings for module
4.12	Packages

4.1 The def statement

The def statement is used to define a function in Python. Functions are a key aspect of Python programming, allowing you to encapsulate code into reusable blocks that perform specific tasks. Here, we'll discuss the def statement in detail, including its syntax, components, and usage.

Syntax

def function_name(parameters):

```
# Function body
statements
return value
```

Components of the def Statement

1. **def Keyword**:
   - The def keyword starts the function definition. It tells Python that a new function is being defined.

2. **Function Name**:
   - The function name is an identifier used to call the function later. It should be descriptive and follow the naming conventions (usually lowercase with words separated by underscores).

3. **Parameters**:
   - Parameters are optional. They are specified within parentheses after the function name and act as placeholders for the values that will be passed to the function. If there are no parameters, you simply leave the parentheses empty.

4. **Colon (:)**:
   - The colon at the end of the function signature indicates the start of the function body.

5. **Function Body**:
   - The function body contains the code that defines what the function does. It is indented to show that it belongs to the function. The body can contain multiple statements, including other function calls, control flow statements, and more.

6. **Return Statement**:
   - The return statement is optional. It allows the function to send back a result to the caller. If no return statement is present, the function returns None by default.

Usage of def Statement

The primary use of the def statement is to create functions. Functions are used to:

- **Organize Code**: Break down complex problems into smaller, manageable pieces.
- **Reuse Code**: Write code once and reuse it multiple times by calling the function.
- **Improve Readability**: Make the code more readable and maintainable by encapsulating functionality.

Example

Here's a simple example to illustrate the def statement:

```
def greet(name):
    print(f"Hello, {name}!")
greet("Alice")  # Output: Hello, Alice!
```

In this example:

- def greet(name): defines a function named greet with a single parameter name.
- The function body contains a single statement that prints a greeting message.

- greet("Alice") calls the function with the argument "Alice", resulting in the output "Hello, Alice!".

Return Statement
The return statement is used to exit a function and return a value to the caller:

```
def add(a, b):
    return a + b
result = add(3, 5)
print(result)  # Output: 8
```

Here, add is a function that takes two parameters a and b, adds them, and returns the result.

Default Parameters
You can define default values for parameters, which are used if no arguments are provided:

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
greet()         # Output: Hello, Guest!
greet("Alice")  # Output: Hello, Alice!
```

In this example, if no argument is passed to greet, it uses the default value "Guest".

Summary
The def statement is a fundamental construct in Python used to define functions. Functions created with def help in organizing code, making it reusable, readable, and maintainable. Understanding how to define and use functions effectively is crucial for writing efficient Python code.

4.2 Returning values

The concept of returning values in Python functions is crucial for creating effective and reusable code. When a function performs a task, it often needs to send the result back to the part of the program that called it. This is done using the return statement. Here's a detailed explanation of returning values in Python functions:

The return Statement
The return statement is used to exit a function and optionally pass an expression back to the caller. The syntax is:

return expression

- **expression**: This is the value or computation that you want to return. It can be any valid Python expression. If the return statement is omitted or the expression is omitted, the function will return None by default.

Basic Example

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

```
result = add(3, 5)
print(result)  # Output: 8
```

In this example:

- The add function takes two parameters, a and b.
- It returns the sum of a and b using the return statement.
- When add(3, 5) is called, the function returns 8, which is stored in the variable result.

Returning Multiple Values

Python functions can return multiple values by using tuples. This is useful when you need to return more than one piece of data from a function.

```
def get_user_info():
    name = "Alice"
    age = 30
    return name, age
user_name, user_age = get_user_info()
print(user_name)  # Output: Alice
print(user_age)   # Output: 30
```

Here, get_user_info returns two values, name and age, as a tuple. These values are then unpacked into the variables user_name and user_age.

Returning a List or Dictionary

Functions can also return complex data structures like lists or dictionaries, allowing you to return multiple related pieces of information.

```
def get_squares(numbers):
    squares = [n ** 2 for n in numbers]
    return squares
nums = [1, 2, 3, 4]
squares_list = get_squares(nums)
print(squares_list)  # Output: [1, 4, 9, 16]
```

In this example, get_squares returns a list of squares of the input numbers.

```
def get_user_profile():
    profile = {
        "name": "Alice",
        "age": 30,
        "email": "alice@example.com"
    }
    return profile
user_profile = get_user_profile()
print(user_profile)
```

Output: {'name': 'Alice', 'age': 30, 'email': 'alice@example.com'}

Here, get_user_profile returns a dictionary containing user profile information.

Conditional Returns
You can use conditional statements within a function to return different values based on some condition.

```
def check_even_odd(number):
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"
result = check_even_odd(4)
print(result)  # Output: Even
```

In this example, check_even_odd returns "Even" if the input number is even and "Odd" if it is odd.

Early Exit with return
The return statement can be used to exit a function early, even before reaching the end of the function body.

```
def find_first_even(numbers):
    for n in numbers:
        if n % 2 == 0:
            return n
    return None
nums = [1, 3, 5, 8, 9]
first_even = find_first_even(nums)
print(first_even)  # Output: 8
```

Here, find_first_even returns the first even number it encounters in the list. If no even number is found, it returns None.

No Explicit Return
If a function does not have an explicit return statement, it returns None by default.

```
def greet(name):
    print(f"Hello, {name}!")
result = greet("Alice")
print(result)  # Output: None
```

In this example, greet prints a message but does not return any value, so result is None.

Summary
The return statement in Python functions allows you to send a result back to the caller. You can return simple values, multiple values using tuples, complex data structures like lists or dictionaries, and use conditional or early returns to control the flow of your function. Understanding how to effectively use the return statement is crucial for writing functional and efficient Python code.

4.3 Parameters

Parameters are a fundamental aspect of functions in Python. They allow you to pass information into functions, enabling the functions to perform operations on different pieces of data without modifying the function's code itself. Here, we'll discuss parameters in detail, including their types, how they work, and their usage.

What Are Parameters?
Parameters are variables that are defined in a function signature. They act as placeholders for the actual values (arguments) that will be passed to the function when it is called. This allows functions to operate on different data inputs dynamically.

Types of Parameters

1. **Positional Parameters**
2. **Default Parameters**
3. **Keyword Parameters**
4. **Variable-Length Parameters**

1. Positional Parameters
Positional parameters are the most basic type. The arguments passed to the function must be in the same order as the parameters defined in the function.

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 30)  # Output: Hello, Alice! You are 30 years old.
```

In this example:

- name and age are positional parameters.
- When calling greet("Alice", 30), "Alice" is assigned to name, and 30 is assigned to age.

2. Default Parameters
Default parameters allow you to define default values for parameters. If no argument is passed for that parameter, the default value is used.

```
def greet(name, age=25):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice")       # Output: Hello, Alice! You are 25 years old.
greet("Bob", 30)     # Output: Hello, Bob! You are 30 years old.
```

In this example:

- age has a default value of 25.
- Calling greet("Alice") uses the default value for age.
- Calling greet("Bob", 30) uses 30 for age.

3. Keyword Parameters
Keyword parameters allow you to pass arguments by explicitly naming them. This makes the function calls more readable and allows arguments to be passed in any order.

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(name="Alice", age=30)  # Output: Hello, Alice! You are 30 years old.
greet(age=30, name="Alice")  # Output: Hello, Alice! You are 30 years old.
```

In this example:
- Arguments are passed using the parameter names name and age.
- The order of arguments does not matter when using keyword parameters.

4. Variable-Length Parameters
Variable-length parameters allow you to pass an arbitrary number of arguments to a function. There are two types:
- ***args**: For non-keyword variable-length arguments.
- ****kwargs**: For keyword variable-length arguments.

Using *args
The *args parameter allows a function to accept any number of positional arguments.

```
def sum_numbers(*args):
    return sum(args)
print(sum_numbers(1, 2, 3))       # Output: 6
print(sum_numbers(4, 5, 6, 7, 8)) # Output: 30
```

In this example:
- *args collects all positional arguments into a tuple.
- The sum function calculates the sum of all elements in args.

Using **kwargs

The **kwargs parameter allows a function to accept any number of keyword arguments.

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_info(name="Alice", age=30, city="New York")

# Output:
# name: Alice
# age: 30
# city: New York
```

In this example:
- **kwargs collects all keyword arguments into a dictionary.
- The function iterates through the dictionary and prints each key-value pair.

Combining Different Types of Parameters
You can combine different types of parameters in a single function. The order should be:

1. Positional parameters
2. Default parameters
3. *args
4. **kwargs

```
def example_func(a, b=2, *args, **kwargs):
   print(f"a: {a}, b: {b}")
   print("args:", args)
   print("kwargs:", kwargs)

example_func(1, 3, 4, 5, 6, x=7, y=8)

# Output:
# a: 1, b: 3
# args: (4, 5, 6)
# kwargs: {'x': 7, 'y': 8}
```

In this example:

- a is a positional parameter.
- b is a default parameter.
- *args collects additional positional arguments.
- **kwargs collects additional keyword arguments.

Summary
Parameters are essential for creating flexible and reusable functions in Python. Understanding the different types of parameters—positional, default, keyword, and variable-length—enables you to write more versatile and powerful functions. By combining these parameter types, you can design functions that handle a wide variety of input scenarios efficiently.

4.4 Arguments

Arguments are the actual values passed to a function when it is called. They provide the input data that the function uses to perform its operations and produce a result. Understanding how to use arguments effectively is crucial for writing flexible and reusable code. Here's a detailed explanation of the concept of arguments in Python, covering their types, how they are passed, and best practices.

Types of Arguments

1. **Positional Arguments**
2. **Keyword Arguments**
3. **Default Arguments**
4. **Variable-Length Arguments**

1. Positional Arguments

Positional arguments are the most basic type of arguments. When you call a function, you provide the values in the same order as the function's parameters.

Example:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 30)  # Output: Hello, Alice! You are 30 years old.
```

In this example:

- "Alice" is passed to the name parameter.
- 30 is passed to the age parameter.
- The order of the arguments matters and must match the order of the parameters.

2. Keyword Arguments

Keyword arguments allow you to pass values to specific parameters by name. This makes the function call more readable and allows you to pass arguments in any order.

Example:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(name="Alice", age=30)  # Output: Hello, Alice! You are 30 years old.
greet(age=30, name="Alice")  # Output: Hello, Alice! You are 30 years old.
```

In this example:

- Arguments are passed using the parameter names name and age.
- The order of arguments does not matter when using keyword arguments.

3. Default Arguments

Default arguments allow you to set default values for parameters. If an argument is not provided for that parameter, the default value is used.

Example:

```
def greet(name, age=25):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice")       # Output: Hello, Alice! You are 25 years old.
greet("Bob", 30)     # Output: Hello, Bob! You are 30 years old.
```

In this example:

- age has a default value of 25.
- If no argument is provided for age, the default value is used.
- If an argument is provided for age, it overrides the default value.

4. Variable-Length Arguments

Variable-length arguments allow a function to accept an arbitrary number of arguments. There are two types:

- ***args**: For non-keyword variable-length arguments.
- ****kwargs**: For keyword variable-length arguments.

Using *args
The *args parameter allows a function to accept any number of positional arguments. These arguments are stored in a tuple.

Example:

```
def sum_numbers(*args):
    return sum(args)

print(sum_numbers(1, 2, 3))       # Output: 6
print(sum_numbers(4, 5, 6, 7, 8)) # Output: 30
```

In this example:

- *args collects all positional arguments into a tuple.
- The sum function calculates the sum of all elements in args.

Using **kwargs
The **kwargs parameter allows a function to accept any number of keyword arguments. These arguments are stored in a dictionary.

Example:

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

print_info(name="Alice", age=30, city="New York")
# Output:
# name: Alice
# age: 30
# city: New York
```

In this example:

- **kwargs collects all keyword arguments into a dictionary.
- The function iterates through the dictionary and prints each key-value pair.

Passing Arguments by Reference or Value
Python's argument passing mechanism is often described as "pass-by-object-reference" or "pass-by-assignment." This means that when you pass arguments to a function, you are passing references to the objects, not the actual objects themselves.

Immutable Objects
For immutable objects (like integers, floats, strings, and tuples), changes to the object within the function do not affect the original object.

Example:

```
def modify_string(s):
    s = s + " World"

my_string = "Hello"
modify_string(my_string)
print(my_string)  # Output: Hello
```

In this example:

- my_string remains unchanged because strings are immutable.

Mutable Objects

For mutable objects (like lists, dictionaries, and sets), changes to the object within the function affect the original object.

Example:

```
def modify_list(lst):
    lst.append(4)

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

In this example:

- my_list is modified because lists are mutable.

Combining Different Types of Arguments

You can combine different types of arguments in a single function. The order should be:

1. Positional arguments
2. Default arguments
3. *args
4. **kwargs

Example:

```
def example_func(a, b=2, *args, **kwargs):
    print(f"a: {a}, b: {b}")
    print("args:", args)
    print("kwargs:", kwargs)

example_func(1, 3, 4, 5, 6, x=7, y=8)
# Output:
# a: 1, b: 3
# args: (4, 5, 6)
# kwargs: {'x': 7, 'y': 8}
```

In this example:

- a is a positional argument.
- b is a default argument.
- *args collects additional positional arguments.
- **kwargs collects additional keyword arguments.

Best Practices for Using Arguments

1. **Use Descriptive Parameter Names**: Choose meaningful names for your parameters to make your code more readable.
2. **Keep Functions Small**: Try to limit the number of parameters a function takes. If a function requires many parameters, consider breaking it into smaller functions.
3. **Use Default Arguments Wisely**: Provide default values for parameters when it makes sense, but avoid using mutable objects as default arguments.
4. **Prefer Keyword Arguments for Clarity**: When a function has many parameters, use keyword arguments to make the function call more readable and clear.
5. **Handle Variable-Length Arguments Carefully**: Ensure that your function correctly handles the case where *args or **kwargs might be empty.

Summary

Arguments are the lifeblood of functions in Python, enabling them to work with dynamic input data. By understanding the different types of arguments—positional, keyword, default, and variable-length—you can write flexible and powerful functions. Additionally, being mindful of how arguments are passed (by reference or value) and following best practices will help you create efficient and maintainable code.

4.5 Local variables

Local variables are an essential concept in Python, and understanding them is crucial for writing effective and bug-free code. This detailed explanation covers what local variables are, how they work, their scope, and best practices for using them.

What Are Local Variables?

Local variables are variables that are declared inside a function and are accessible only within that function. They exist only during the execution of the function and are destroyed once the function call is complete.

Key Characteristics of Local Variables

1. **Scope**: Local variables are only accessible within the function in which they are declared.
2. **Lifetime**: They are created when the function is called and destroyed when the function exits.
3. **Isolation**: Each function call has its own set of local variables, which are independent of those in other function calls.

Declaring and Using Local Variables

Local variables are defined within a function's body. Here's an example to illustrate their declaration and usage:

```
def example_function():
    local_var = 10  # local variable
    print(local_var)

example_function()  # Output: 10
print(local_var)    # Error: NameError: name 'local_var' is not defined
```

In this example:

- local_var is a local variable, declared within example_function.
- Attempting to print local_var outside the function results in an error because it is not accessible outside its defining function.

Scope of Local Variables

The scope of a local variable is limited to the block of code in which it is declared. This means it can only be used within the function where it is defined.

Example:

```
def outer_function():
    outer_var = "I am outer"

    def inner_function():
        inner_var = "I am inner"
        print(outer_var)  # Accessible
        print(inner_var)  # Accessible

    inner_function()
    print(outer_var)      # Accessible
    print(inner_var)      # Error: NameError: name 'inner_var' is not defined

outer_function()
```

In this example:

- outer_var is a local variable of outer_function and is accessible within both outer_function and inner_function.
- inner_var is a local variable of inner_function and is not accessible outside of it.

Local Variables vs. Global Variables

Global variables are variables declared outside of any function and are accessible throughout the module. Local variables, on the other hand, are limited to the function they are defined in.

Example:

```
global_var = "I am global"

def example_function():
    local_var = "I am local"
    print(global_var)  # Accessible
    print(local_var)   # Accessible
```

```
example_function()
print(global_var)      # Accessible
print(local_var)       # Error: NameError: name 'local_var' is not defined
```

In this example:

- global_var is a global variable and is accessible both inside and outside the function.
- local_var is a local variable and is only accessible within example_function.

The global Keyword

Sometimes you may need to modify a global variable within a function. This can be done using the global keyword.

Example:

```
global_var = "I am global"

def modify_global():
    global global_var
    global_var = "I am modified"

modify_global()
print(global_var)  # Output: I am modified
```

In this example:

- The global keyword is used to declare that global_var inside the function refers to the global variable global_var.
- The modification to global_var inside modify_global affects the global variable.

The nonlocal Keyword

The nonlocal keyword allows you to modify a variable in an enclosing (but non-global) scope, such as in nested functions.

Example:

```
def outer_function():
    outer_var = "I am outer"

    def inner_function():
        nonlocal outer_var
        outer_var = "I am modified in inner"

    inner_function()
    print(outer_var)  # Output: I am modified in inner

outer_function()
```

In this example:

- The nonlocal keyword is used to declare that outer_var inside inner_function refers to the outer_var in the enclosing scope of outer_function.

- The modification to outer_var inside inner_function affects the variable in outer_function.

Best Practices for Local Variables

1. **Use Descriptive Names**: Choose meaningful names for your local variables to make your code more readable.
2. **Limit Scope**: Declare variables in the narrowest scope possible to reduce the risk of unintended side effects.
3. **Avoid Global Variables**: Minimize the use of global variables as they can lead to code that is difficult to understand and debug.
4. **Use Constants**: For values that should not change, consider using constants to avoid accidental modifications.

Summary

Local variables are a fundamental concept in Python, providing a way to store and manipulate data within a function. Understanding their scope, lifetime, and how they differ from global variables is essential for writing clean, efficient, and bug-free code. By following best practices and using keywords like global and nonlocal when appropriate, you can effectively manage the variables in your Python programs.

4.6 Global variables and the global statement

Understanding global variables and the global statement is crucial for managing data and state across different parts of your Python program. This detailed explanation covers what global variables are, how to use the global statement, and best practices for working with global variables.

What Are Global Variables?

Global variables are variables declared outside any function, and they are accessible throughout the entire module. They retain their value throughout the program's execution and can be read from any function.

Key Characteristics of Global Variables

1. **Scope**: Global variables are accessible from any part of the code, including inside functions (unless shadowed by local variables).
2. **Lifetime**: They exist for the duration of the program's execution.
3. **Visibility**: They can be accessed by any function or code block within the same module.

Declaring and Using Global Variables

Global variables are typically declared at the top of the module, outside of any function definitions. Here's an example to illustrate the declaration and usage of global variables:

```
global_var = "I am global"

def example_function():
   print(global_var)  # Accessible

example_function()      # Output: I am global
```

```
print(global_var)       # Output: I am global
```

In this example:

- global_var is declared outside any function, making it a global variable.
- It can be accessed both inside example_function and outside it.

Modifying Global Variables

By default, functions cannot modify global variables directly; they can only read them. To modify a global variable within a function, you need to use the global keyword.

Example without global Keyword:

```
global_var = "I am global"

def modify_global():
    global_var = "I am modified"  # This creates a new local variable, doesn't modify the global one

modify_global()
print(global_var)  # Output: I am global
```

In this example:

- A local variable global_var is created within modify_global, which shadows the global variable.
- The global variable global_var remains unchanged.

Using the global Keyword

The global keyword allows you to declare that a variable inside a function refers to a global variable. This enables you to modify the global variable from within the function.

Example with global Keyword:

```
global_var = "I am global"

def modify_global():
    global global_var
    global_var = "I am modified"

modify_global()
print(global_var)  # Output: I am modified
```

In this example:

- The global keyword is used to declare that global_var inside modify_global refers to the global variable.
- The modification to global_var inside modify_global affects the global variable.

Accessing Global Variables from Nested Functions

You can access global variables from within nested functions, but modifying them requires the global keyword in the enclosing function as well.

Example:

```
global_var = "I am global"

def outer_function():
    global global_var
    def inner_function():
        global_var = "I am modified in inner"
    inner_function()

outer_function()
print(global_var)  # Output: I am modified in inner
```

In this example:

- The global keyword in outer_function allows inner_function to modify the global variable.

Best Practices for Using Global Variables

1. **Minimize Use**: Use global variables sparingly to avoid potential conflicts and unintended side effects.
2. **Use Constants for Immutable Values**: If the global variable should not change, use all uppercase names to indicate constants (e.g., PI = 3.14).
3. **Encapsulation**: Encapsulate global variables within classes or modules to limit their scope and improve code organization.
4. **Clear Naming**: Use descriptive names for global variables to make their purpose clear and avoid naming conflicts.
5. **Documentation**: Document the use and purpose of global variables to improve code readability and maintainability.

Summary

Global variables are accessible throughout your entire module and retain their value for the duration of the program's execution. The global keyword allows functions to modify these global variables. While global variables can be useful, they should be used judiciously to avoid potential issues with code readability and maintainability. Following best practices for naming, encapsulation, and documentation can help you manage global variables effectively in your Python programs.

4.7 Doc strings for functions

Docstrings, or documentation strings, are a critical feature in Python used to document modules, classes, functions, and methods. They provide a convenient way to associate documentation with Python code. Here's a detailed explanation of docstrings, focusing on their use in functions, their importance, and best practices.

What Are Docstrings?

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It is used to document the purpose and usage of the specific segment of code it is associated with. In Python, docstrings are denoted by triple quotes (""" or ''').

Importance of Docstrings

1. **Readability**: Docstrings improve code readability by providing inline documentation.
2. **Maintenance**: They make it easier to understand and maintain code, especially for complex functions.
3. **Auto-Documentation**: Tools like Sphinx can automatically generate documentation from docstrings.
4. **Interactive Help**: Docstrings can be accessed using the help() function in interactive environments.

Writing Docstrings for Functions
To write a docstring for a function, place the string literal immediately after the function header. Here's a basic example:

```
def example_function(param1, param2):
    """
    This is an example function.

    Parameters:
    param1 (int): The first parameter.
    param2 (str): The second parameter.

    Returns:
    bool: The return value. True for success, False otherwise.
    """
    # Function body
    return True
```

Accessing Docstrings
You can access a function's docstring using the __doc__ attribute or the help() function.

```
print(example_function.__doc__)
help(example_function)
```

Components of a Good Docstring
A well-written docstring typically includes the following components:

1. **Summary**: A brief description of the function's purpose.
2. **Parameters**: A description of the function's parameters, including their types and meanings.
3. **Returns**: A description of the return value, including its type.
4. **Raises**: A description of any exceptions raised by the function (optional).

Detailed Example
Here's a more detailed example with a function that includes all these components:

```
def divide_numbers(numerator, denominator):
    """
    Divide two numbers and return the result.

    Parameters:
    numerator (float): The numerator of the division.
    denominator (float): The denominator of the division. Must not be zero.
```

```
    Returns:
    float: The result of the division.

    Raises:
    ValueError: If the denominator is zero.
    """
    if denominator == 0:
        raise ValueError("Denominator must not be zero.")
    return numerator / denominator
```

Explanation of the Components

- **Summary**: "Divide two numbers and return the result."
- **Parameters**:
    - numerator (float): The numerator of the division.
    - denominator (float): The denominator of the division. Must not be zero.
- **Returns**: float: The result of the division.
- **Raises**: ValueError: If the denominator is zero.

Best Practices for Writing Docstrings

1. **Use Triple Quotes**: Always use triple quotes for docstrings, even for one-liners, to maintain consistency.
2. **Keep It Simple and Clear**: Write clear and concise descriptions. Avoid unnecessary complexity.
3. **Use Proper Formatting**: Use proper indentation and formatting to enhance readability. Follow conventions like PEP 257.
4. **Be Descriptive**: Clearly describe what the function does, its parameters, return values, and any exceptions it raises.
5. **Stay Updated**: Keep docstrings updated as the function's implementation changes.

Example Following PEP 257 Conventions

PEP 257 is the Python Enhancement Proposal that provides conventions for writing docstrings. Here's an example that follows these conventions:

```
def greet(name):
    """
    Greet a person by their name.

    This function prints a greeting message that includes the provided name.

    Parameters:
    name (str): The name of the person to greet.

    Returns:
    None
    """
    print(f"Hello, {name}!")
```

Summary

Docstrings are an essential part of writing clear, maintainable, and well-documented Python code. They provide a straightforward way to describe the purpose, parameters, return values, and exceptions of functions. By following best practices and conventions, you can create informative docstrings that enhance the readability and usability of your code.

4.8 Decorators for functions

Decorators are a powerful and versatile feature in Python that allows you to modify or enhance the behavior of functions or methods without permanently modifying their code. This detailed explanation covers what decorators are, how they work, their syntax, and various use cases.

What Are Decorators?
Decorators are functions that take another function as an argument, extend or alter its behavior, and return a new function with the enhanced behavior. They provide a clean and readable way to apply the same behavior across multiple functions or methods.

Basic Syntax of Decorators
A decorator is applied to a function using the @decorator_name syntax directly above the function definition.

```
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        # Add functionality before calling the original function
        result = original_function(*args, **kwargs)
        # Add functionality after calling the original function
        return result
    return wrapper_function

@decorator_function
def display():
    print("Display function executed")
display()
```

In this example:

- decorator_function is the decorator.
- wrapper_function is the inner function that adds additional behavior before and after calling original_function.
- @decorator_function applies the decorator to display.

Detailed Example
Here's a more detailed example illustrating how decorators work:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
```

```
@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:
Something is happening before the function is called.
Hello!
Something is happening after the function is called.

In this example:

- my_decorator is a decorator that prints messages before and after calling the original function say_hello.
- @my_decorator applies this behavior to say_hello.

Decorators with Arguments
Decorators can also take arguments if you need to pass specific values to the decorator.

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Output:
Hello, Alice!
Hello, Alice!
Hello, Alice!

In this example:

- repeat is a decorator factory that takes an argument num_times and returns a decorator.
- @repeat(num_times=3) applies this decorator to greet, causing it to repeat its execution three times.

Chaining Decorators
Multiple decorators can be applied to a single function. The decorators are applied from the bottom up.
python
Copy code
```
def decorator1(func):
```

```
    def wrapper():
        print("Decorator 1")
        func()
    return wrapper

def decorator2(func):
    def wrapper():
        print("Decorator 2")
        func()
    return wrapper

@decorator1
@decorator2
def greet():
    print("Hello!")

greet()
```

Output:
Decorator 1
Decorator 2
Hello!

In this example:

- @decorator2 is applied first, then @decorator1.
- The output reflects the order of execution.

Using functools.wraps
When you create decorators, it's a good practice to use functools.wraps to preserve the original function's metadata, such as its name and docstring.

```
import functools

def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

@my_decorator
def say_hello():
    """This function says hello."""
    print("Hello!")

print(say_hello.__name__)
print(say_hello.__doc__)
```

Output:
say_hello
This function says hello.

In this example:

- functools.wraps is used to copy the metadata from func to wrapper, preserving the original function's name and docstring.

Practical Use Cases

Logging
A common use of decorators is to add logging to functions.

```
def log_function_call(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args} and {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

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

add(3, 4)
```

Output:
Calling add with (3, 4) and {}
add returned 7

Authorization
Decorators can also be used to check user permissions.

```
def requires_permission(permission):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            user_permissions = get_current_user_permissions()
            if permission in user_permissions:
                return func(*args, **kwargs)
            else:
                raise PermissionError("You do not have permission to access this resource.")
        return wrapper
    return decorator

@requires_permission('admin')
def delete_user(user_id):
    print(f"User {user_id} deleted")
```

```
delete_user(123)
```

In this example:

- requires_permission checks if the current user has the required permission before calling the function.

Summary

Decorators in Python are a powerful tool for modifying or enhancing the behavior of functions and methods. They provide a clean and readable way to apply common behavior across multiple functions. By understanding the basic syntax, how to pass arguments, chain multiple decorators, and use functools.wraps, you can leverage decorators effectively in your Python code. They are particularly useful for logging, authorization, caching, and other cross-cutting concerns.

Creating a custom decorator

Creating a custom decorator in Python involves defining a function that takes another function as an argument, modifies its behavior, and returns a new function. This allows you to add or alter functionality in a reusable and modular way. Here's a detailed guide on how to create and use custom decorators.

Basic Structure of a Decorator

1. **Define the Outer Function**: This function will take the original function as an argument.
2. **Define the Wrapper Function**: Inside the outer function, define a new function that will add or modify behavior. This function will call the original function and can also include code to execute before or after the call.
3. **Return the Wrapper Function**: The outer function should return the wrapper function.

Basic Example

Here's a simple example of a custom decorator that prints messages before and after calling a function:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Something is happening before the function is called.
```

Hello!
Something is happening after the function is called.

Detailed Explanation

1. **Outer Function (my_decorator)**:
    - Takes func as an argument, which is the function to be decorated.
2. **Inner Function (wrapper)**:
    - Adds extra functionality before and after the call to func.
    - Calls func() inside it.
3. **Apply the Decorator (@my_decorator)**:
    - Decorates say_hello with my_decorator.
    - When say_hello is called, it actually calls wrapper.

Decorators with Arguments

You can create decorators that accept arguments by adding another level of function nesting.

Example with Arguments

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                func(*args, **kwargs)
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Output:

```
Hello, Alice!
Hello, Alice!
Hello, Alice!
```

Explanation:

1. **Outer Function (repeat)**:
    - Takes num_times as an argument and returns decorator_repeat.
2. **Decorator Function (decorator_repeat)**:
    - Takes the original function func and returns the wrapper function.
3. **Wrapper Function**:
    - Repeats the execution of func according to num_times.

Preserving Metadata with functools.wraps

When creating decorators, it’s a good practice to use functools.wraps to ensure that the metadata of the original function (like its name and docstring) is preserved.

Example with functools.wraps

```
import functools

def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

@my_decorator
def say_hello(name):
    """This function says hello."""
    print(f"Hello, {name}!")

print(say_hello.__name__)  # Output: say_hello
print(say_hello.__doc__)   # Output: This function says hello.
```

Explanation:

- functools.wraps(func) copies the metadata from func to wrapper.

Decorators with Arguments and Metadata Preservation
You can combine decorators with arguments and metadata preservation using functools.wraps.

```
import functools

def repeat(num_times):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                func(*args, **kwargs)
        return wrapper
    return decorator_repeat

@repeat(num_times=2)
def greet(name):
    """This function greets a person."""
    print(f"Hello, {name}!")

print(greet.__name__)  # Output: greet
print(greet.__doc__)   # Output: This function greets a person.
```

Explanation:

- functools.wraps is used inside decorator_repeat to preserve greet's metadata.

Summary
To create a custom decorator in Python:
1. Define an outer function that takes the function to be decorated.
2. Define an inner wrapper function that adds functionality and calls the original function.
3. Return the wrapper function from the outer function.
4. Optionally, use functools.wraps to preserve the original function's metadata.

Decorators are a powerful tool for modifying function behavior and can be combined with arguments for more flexibility. If you have any more questions or need further examples, feel free to

4.9 Lambda Iterators and generators

Definition
Lambda functions, also known as anonymous functions, are a way to create small, throwaway functions without formally defining them with the def keyword. They are useful for simple operations that can be encapsulated in a single expression.

Syntax
lambda arguments: expression

- **lambda**: Keyword used to define the lambda function.
- **arguments**: A comma-separated list of parameters (similar to function arguments).
- **expression**: A single expression that is evaluated and returned.

Characteristics
1. **Anonymous**: Lambda functions do not have a name unless assigned to a variable.
2. **Single Expression**: They are limited to a single expression and cannot include statements or multiple expressions.
3. **Implicit Return**: The expression's result is automatically returned.

Examples

- **Basic Lambda**

```
# Lambda function to add two numbers
add = lambda x, y: x + y

print(add(5, 3))  # Output: 8
```

- **Using Lambda with map()**

map() applies a function to all items in an iterable.

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x**2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

- **Using Lambda with filter()**

filter() filters items based on a function that returns True or False.

```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4]
```

Using Lambda with sorted()
You can sort items based on a custom key.

```
data = [(1, 'apple'), (2, 'banana'), (3, 'cherry')]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)  # Output: [(1, 'apple'), (2, 'banana'), (3, 'cherry')]
```

- **Advanced Lambda Uses**

Inline Lambda with List Comprehension

```
numbers = [1, 2, 3, 4]
doubled = [(lambda x: x*2)(x) for x in numbers]
print(doubled)  # Output: [2, 4, 6, 8]
```

Limitations of Lambda Functions

1. **Limited to Single Expression**: Lambda functions cannot contain multiple statements or complex logic.
2. **Less Readable**: For more complex operations, lambda functions can become hard to read and maintain.

Iterators
Definition
An iterator is an object that represents a stream of data. It is a sequence object that implements the iterator protocol, consisting of __iter__() and __next__() methods.
Iterator Protocol

1. **__iter__()**: Returns the iterator object itself. This is required for the object to be used in a for loop or other iteration contexts.
2. **__next__()**: Returns the next item from the sequence. When there are no more items, it raises the StopIteration exception.

Creating an Iterator
Using a Class
python
Copy code

```
class MyIterator:
    def __init__(self, start, end):
```

```
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
            raise StopIteration
        value = self.current
        self.current += 1
        return value

it = MyIterator(1, 3)
for num in it:
    print(num)
```

Explanation:

- __init__() initializes the start and end values.
- __iter__() returns the iterator object.
- __next__() generates the next item and increments the counter, raising StopIteration when done.

Common Iterator Methods

- **iter()**: Converts an iterable into an iterator.
- **next()**: Retrieves the next item from an iterator.

```
numbers = [1, 2, 3]
iterator = iter(numbers)
print(next(iterator))  # Output: 1
print(next(iterator))  # Output: 2
```

Iterators vs. Iterables

- **Iterable**: Any object that has an __iter__() method and can return an iterator (e.g., lists, tuples).
- **Iterator**: An object that implements both __iter__() and __next__().

Common Uses

1. **Traversing Collections**: Iterators are used to loop through elements in data structures.
2. **Lazy Evaluation**: They are useful for generating items on-the-fly, which is memory efficient.

Generators

Definition

Generators are a type of iterator created using functions with yield statements. They simplify iterator creation and provide a more readable way to handle iteration.

Syntax

```
def generator_function():
    yield value
```

- **def**: Defines a generator function.
- **yield**: Suspends the function's state and returns the value, resuming from where it left off when next() is called.

Characteristics

1. **Stateful**: Generators maintain their state between yield calls.
2. **Lazy Evaluation**: They generate values one at a time, only when needed, which is efficient for large datasets.

Example

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

gen = count_up_to(3)
for number in gen:
    print(number)
```

Explanation:

- count_up_to yields numbers from 1 to max.
- Each call to next() resumes the function from the last yield statement.

Generator Expressions

Generator expressions are a concise way to create generators using a syntax similar to list comprehensions.

```
squared_gen = (x**2 for x in range(1, 4))
for square in squared_gen:
    print(square)
```

Explanation:

- (x**2 for x in range(1, 4)) creates a generator that yields the squares of numbers from 1 to 3.

Comparison with Lists

- **Memory Usage**: Generators are more memory-efficient as they do not store all items at once.
- **Performance**: Generators are faster for large datasets as they produce values on-the-fly.

Common Uses

1. **Processing Large Datasets**: Ideal for handling large streams of data without consuming large amounts of memory.
2. **Pipelines**: Useful for chaining operations in a data processing pipeline.

Advanced Generator Features

1. **Sending Values**: Generators can receive values via the send() method.

```
def echo():
   while True:
      received = yield
      print(f"Received: {received}")

gen = echo()
next(gen)  # Prime the generator
gen.send("Hello")  # Output: Received: Hello
```

2. **Generator States**: Generators can be paused and resumed, and can handle complex stateful processing.

Summary

- **Lambda Functions**: Provide a concise way to define simple, anonymous functions used in functional programming contexts.
- **Iterators**: Objects that implement the __iter__() and __next__() methods for traversing sequences.
- **Generators**: Special types of iterators created with yield, offering a more readable and memory-efficient way to handle sequences and lazy evaluation.

4.10 Modules

A module is a file containing Python code. It can define functions, classes, and variables, and can also include runnable code. Modules help in organizing code logically by grouping related functionality into separate files.

Why Use Modules?

1. **Code Reusability**: Once a module is written, it can be reused in other programs.
2. **Code Organization**: Modules help in organizing code into logical units, making it easier to understand and maintain.
3. **Namespace Management**: Modules provide a namespace that helps in avoiding name clashes.

- **Creating and Using Modules**

Creating a Module

To create a module, simply write Python code in a file with a .py extension.

Example: math_utils.py

```
# math_utils.py
```

```
def add(x, y):
    return x + y
def subtract(x, y):
    return x - y
```

Using a Module
You can use the import statement to include a module in your program.

Example:

```
# main.py
import math_utils
result = math_utils.add(5, 3)
print(result)  # Output: 8
```

Importing Specific Functions
You can import specific functions from a module.

```
# main.py
from math_utils import add
result = add(5, 3)
print(result)  # Output: 8
```

Importing All Functions
You can import all functions from a module using the * wildcard. However, this is generally discouraged as it can lead to ambiguity.

```
# main.py

from math_utils import *
result = subtract(10, 5)
print(result)  # Output: 5
```

Aliasing Modules and Functions
You can give a module or function an alias using the as keyword to shorten the reference.

```
# main.py
import math_utils as mu
result = mu.add(5, 3)
print(result)  # Output: 8
```

Python Standard Library Modules
Python comes with a rich set of standard library modules that provide a wide range of functionalities. Some commonly used standard modules include:

1. **math**: Provides mathematical functions.

```
import math
print(math.sqrt(16))  # Output: 4.0
```

2. **os**: Provides functions for interacting with the operating system.

```
import os
```

```
print(os.getcwd())  # Output: Current working directory
```

3. **sys**: Provides access to system-specific parameters and functions.

```
import sys
print(sys.version)  # Output: Python version
```

4. **datetime**: Provides functions for manipulating dates and times.

```
from datetime import datetime
now = datetime.now()
print(now)  # Output: Current date and time
```

5. **json**: Provides functions for parsing JSON data.

```
import json
data = json.dumps({"name": "Alice", "age": 30})
print(data)  # Output: {"name": "Alice", "age": 30}
```

Import Mechanisms

Absolute Import
This is the most common form of import where you specify the full path to the module.

```
from package.module import function
```

Relative Import
Relative imports use a relative path to import modules within the same package. They are often used in larger projects.

```
from .module import function  # Relative to the current package
from ..subpackage.module import function  # Relative to the parent package
```

Importing Modules Dynamically
You can import modules dynamically using the importlib module.

```
import importlib
math_utils = importlib.import_module('math_utils')
result = math_utils.add(5, 3)
print(result)  # Output: 8
```

Module Search Path
When importing a module, Python searches for it in the following locations:

1. **The directory containing the input script**.
2. **The directories listed in the PYTHONPATH environment variable**.
3. **The default installation-dependent directory**.

You can view the module search path using the sys.path variable.

```
import sys
print(sys.path)
```

__init__.py and Package Initialization

__init__.py

- **Purpose**: Marks a directory as a Python package. It can also execute initialization code for the package.
- **Contents**: Can be empty or contain package initialization code.

Example:
# mypackage/__init__.py
print("Initializing mypackage")
When you import mypackage, the __init__.py file is executed.

Package Structure
A package is a directory containing modules and a special __init__.py file.
markdown
mypackage/
  __init__.py
  module1.py
  module2.py

Importing from a Package:
from mypackage import module1
from mypackage.module2 import function

Reloading Modules
You can reload a module to re-import it and apply any changes made.
import importlib
import math_utils
importlib.reload(math_utils)

Summary

- **Modules**: Files containing Python code that can define functions, classes, and variables. They help in organizing and reusing code.
- **Creating Modules**: Defined as .py files and used with the import statement.
- **Using Modules**: Import modules using import, from ... import, or import ... as.
- **Standard Library**: Python provides a wide range of built-in modules for various functionalities.
- **Import Mechanisms**: Includes absolute, relative, and dynamic imports.
- **Module Search Path**: Python searches for modules in specific directories.
- **Packages**: Directories containing multiple modules and an __init__.py file.
- **Reloading Modules**: Allows re-importing modules to apply changes.

Commonly Used Built-in Modules

1. os
The os module provides a way to interact with the operating system, including file and directory operations, environment variables, and process management.

Example Uses:

- Get the current working directory:

```
import os
print(os.getcwd())
```

- List files in a directory:

```
import os
print(os.listdir('.'))
```

- Create a new directory:

```
import os
os.mkdir('new_directory')
```

2. sys

The sys module provides access to system-specific parameters and functions. It's commonly used to manipulate the Python runtime environment.

Example Uses:

- Get the Python version:

```
import sys
print(sys.version)
```

- Modify the module search path:

```
import sys
sys.path.append('/my/custom/path')
```

- Exit the program:

```
import sys
sys.exit(0)
```

3. math

The math module provides mathematical functions and constants, such as trigonometric functions, logarithms, and constants like pi.

Example Uses:

- Compute the square root of a number:

```
import math
print(math.sqrt(16))
```

- Calculate the factorial:

```
import math
print(math.factorial(5))
```

- Get the value of pi:

```
import math
print(math.pi)
```

4. datetime

The datetime module supplies classes for manipulating dates and times.

Example Uses:

- Get the current date and time:

```
from datetime import datetime
now = datetime.now()
print(now)
```

- Create a specific date:

```
from datetime import date
d = date(2024, 7, 29)
print(d)
```

- Calculate the difference between two dates:

```
from datetime import date
today = date.today()
future_date = date(2025, 1, 1)
delta = future_date - today
print(delta.days)
```

5. json

The json module provides functions to work with JSON data, including parsing JSON and converting Python objects to JSON.

Example Uses:

- Convert a Python object to a JSON string:

```
import json
data = {'name': 'Alice', 'age': 30}
json_str = json.dumps(data)
print(json_str)
```

- Parse a JSON string into a Python object:

```
import json
json_str = '{"name": "Alice", "age": 30}'
data = json.loads(json_str)
print(data)
```

6. random

The random module provides functions to generate random numbers and perform random operations.

Example Uses:

- Generate a random integer:

```
import random
print(random.randint(1, 10))
```

- Select a random element from a list:

```
import random
items = ['apple', 'banana', 'cherry']
print(random.choice(items))
```

- Shuffle a list:

```
import random
```

```
items = [1, 2, 3, 4, 5]
random.shuffle(items)
print(items)
```

7. re

The re module provides functions for working with regular expressions, allowing complex string searches and manipulations.

Example Uses:

- Search for a pattern in a string:

```
import re
pattern = r'\d+'
text = 'There are 2 apples and 3 oranges.'
matches = re.findall(pattern, text)
print(matches)  # Output: ['2', '3']
```

- Replace text using a pattern:

```
import re
text = 'The price is 45 dollars.'
new_text = re.sub(r'\d+', 'XX', text)
print(new_text)  # Output: The price is XX dollars.
```

8. collections

The collections module provides alternatives to Python's built-in data types, such as named tuples, deques, and counters.

Example Uses:

- Create a named tuple:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(p.x, p.y)
```

- Use a deque (double-ended queue):

```
from collections import deque
d = deque([1, 2, 3])
d.appendleft(0)
d.append(4)
print(d)  # Output: deque([0, 1, 2, 3, 4])
```

- Count occurrences of elements:

```
from collections import Counter
data = ['apple', 'banana', 'apple', 'orange']
counter = Counter(data)
print(counter)  # Output: Counter({'apple': 2, 'banana': 1, 'orange': 1})
```

9. itertools

The itertools module provides functions that create iterators for efficient looping.

Example Uses:

- Create an infinite counter:

```
import itertools
counter = itertools.count(start=10, step=2)
print(next(counter))  # Output: 10
print(next(counter))  # Output: 12
```

- Generate permutations:

```
import itertools
perms = itertools.permutations('ABC')
for perm in perms:
    print(perm)
```

- Create combinations:

```
import itertools
combs = itertools.combinations('ABCD', 2)
for comb in combs:
    print(comb)
```

10. functools

The functools module provides higher-order functions that act on or return other functions.

Example Uses:

- Memoization with lru_cache:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
print(fibonacci(10))  # Output: 55
```

- Partial function application with partial:

```
from functools import partial
def multiply(x, y):
    return x * y
double = partial(multiply, y=2)
print(double(5))  # Output: 10
```

11. csv

The csv module provides tools for reading and writing CSV (Comma Separated Values) files.

Example Uses:

- Reading from a CSV file:

```
import csv
with open('data.csv', mode='r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

- Writing to a CSV file:

```
import csv
data = [['Name', 'Age'], ['Alice', 30], ['Bob', 25]]
with open('data.csv', mode='w', newline='') as file:
   writer = csv.writer(file)
   writer.writerows(data)
```

12. socket

The socket module provides low-level networking interfaces, including creating and using network sockets.

Example Uses:

- Create a simple TCP client:

```
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('localhost', 12345))
client.sendall(b'Hello, server')
response = client.recv(1024)
print('Received:', response.decode())
client.close()
```

- Create a simple TCP server:

```
import socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost', 12345))
server.listen(1)
conn, addr = server.accept()
print('Connected by', addr)
data = conn.recv(1024)
conn.sendall(b'Hello, client')
conn.close()
```

4.11 Doc strings for modules

Module docstrings are string literals that appear right after the module's docstring and before any code in the module file. They provide an overview of the module’s purpose, functionality, and usage. They are defined using triple quotes (""" or '''), which allows for multi-line documentation.

Placement

The module docstring should be the very first statement in the module file. It is placed immediately after the import statements (if any) and before any other code.

Example:

```
"""
module_name.py - A brief description of what this module does
```

This module provides functionalities for performing specific tasks, such as
data processing, calculations, or file operations. It includes the following
functions/classes:

- function1: Description of function1
- class1: Description of class1

Example usage:
    from module_name import function1
    result = function1(arg1, arg2)
    print(result)

Dependencies:
    - Some external libraries
    - Python version compatibility
"""

Components of a Module Docstring
A well-written module docstring typically includes several key components:

1. Module Overview
Provide a high-level description of what the module does. Explain its primary purpose and the problem it solves.

Example:
"""
math_utils.py - Utility functions for basic mathematical operations

This module provides functions for performing common mathematical operations
such as addition, subtraction, multiplication, and division.
"""

2. Function and Class Descriptions
List and describe the functions and classes provided by the module. Mention the key functionality and purpose of each.

Example:
"""
math_utils.py - Utility functions for basic mathematical operations

Functions:
- add(x, y): Returns the sum of x and y.
- subtract(x, y): Returns the difference between x and y.
- multiply(x, y): Returns the product of x and y.
- divide(x, y): Returns the quotient of x and y. Raises ValueError if y is zero.
"""

3. Usage Examples
Provide example code snippets that demonstrate how to use the module. This helps users quickly understand how to apply the module's functionality in their own code.

Example:

```
"""
math_utils.py - Utility functions for basic mathematical operations

Example usage:
    import math_utils

    result = math_utils.add(5, 3)
    print(result)  # Output: 8

    result = math_utils.divide(10, 2)
    print(result)  # Output: 5.0
"""
```

4. Dependencies and Requirements

Mention any external libraries or specific Python versions required by the module. This helps users understand any prerequisites for using the module.

Example:

```
"""
math_utils.py - Utility functions for basic mathematical operations

Dependencies:
    - None
    - Compatible with Python 3.x
"""
```

5. Additional Information

Include any additional information that might be useful, such as known issues, limitations, or references to related modules or documentation.

Example:

```
"""
math_utils.py - Utility functions for basic mathematical operations

Known Issues:
    - The divide function will raise a ValueError if the denominator is zero.
"""
```

Accessing Module Docstrings

You can access a module's docstring using the __doc__ attribute. This can be useful for documentation generation or interactive exploration.

Example:

```
import math_utils

# Accessing the module docstring
```

```
print(math_utils.__doc__)
```

Output:

```
math_utils.py - Utility functions for basic mathematical operations

Functions:
- add(x, y): Returns the sum of x and y.
- subtract(x, y): Returns the difference between x and y.
- multiply(x, y): Returns the product of x and y.
- divide(x, y): Returns the quotient of x and y. Raises ValueError if y is zero.

Example usage:
    import math_utils

    result = math_utils.add(5, 3)
    print(result)  # Output: 8

    result = math_utils.divide(10, 2)
    print(result)  # Output: 5.0
"""
```

Best Practices for Writing Module Docstrings

1. **Be Clear and Concise**: Provide a clear overview of the module's purpose and functionality without unnecessary detail.
2. **Include Examples**: Demonstrate how to use the module with practical examples.
3. **Document Dependencies**: Mention any required external libraries or specific Python versions.
4. **Update as Needed**: Keep the docstring updated as the module evolves.

PEP 257 Conventions

PEP 257 provides conventions for writing docstrings in Python. Some key points include:

- **Summary Line**: Start with a one-line summary of the module's purpose.
- **Blank Line**: Follow the summary line with a blank line.
- **Detailed Description**: Provide additional details after the blank line if necessary.

Example Following PEP 257:

```
"""
math_utils.py - A module for basic mathematical operations

This module provides utility functions for performing basic arithmetic operations,
including addition, subtraction, multiplication, and division.

Functions:
- add(x, y): Returns the sum of x and y.
- subtract(x, y): Returns the difference between x and y.
- multiply(x, y): Returns the product of x and y.
- divide(x, y): Returns the quotient of x and y, raising a ValueError if y is zero.

Example:
    >>> import math_utils
```

```
    >>> math_utils.add(2, 3)
    5
"""
```

Summary

- **Module Docstrings**: Provide documentation at the beginning of a module file, explaining its purpose, functions, usage, and dependencies.
- **Components**: Overview, function/class descriptions, usage examples, dependencies, and additional information.
- **Access**: Use the __doc__ attribute to retrieve module docstrings.
- **Best Practices**: Be clear, include examples, document dependencies, and update regularly.

PROGRAMS

Program to Demonstrate a Function with and without Arguments

```
1. def function_definition_with_no_argument():
2. print("This is a function definition with NO Argument")
3. def function_definition_with_one_argument(message):
4. print(f"This is a function definition with {message}")
5. def main():
6. function_definition_with_no_argument()
7. function_definition_with_one_argument("One Argument")
8. if __name__ == "__main__":
9. main()
```

Output

```
This is a function definition with NO Argument
This is a function definition with One Argument
```

Program to Find the Area of Trapezium Using the Formula Area = (1/2) * (a + b) * h Where a and b Are the 2 Bases of Trapezium and h Is the Height

```
1. def area_trapezium(a, b, h):
2. area = 0.5 * (a + b) * h
3. print(f"Area of a Trapezium is {area}")
4. def main():
5. area_trapezium(10, 15, 20)
6. if __name__ == "__main__":
7. main()
```

Output

```
Area of a Trapezium is 250.0
```

Program to Demonstrate Using the Same Variable Name in Calling Function and Function Definition

```
1. god_name = input("Who is the God of Seas according to Greek Mythology?")
2. def greek_mythology(god_name):
3. print(f"The God of seas according to Greek Mythology is {god_name}")
4. def main():
5. greek_mythology(god_name)
6. if __name__ == "__main__":
7. main()
```

Output

Who is the God of Seas according to Greek Mythology? Poseidon
The God of seas according to Greek Mythology is Poseidon

Program 4.4: Program to Demonstrate the Return of Multiple Values from a Function Definition

```
1. def world_war():
2. alliance_world_war = input("Which alliance won World War 2?")
3. world_war_end_year = input("When did World War 2 end?")
4. return alliance_world_war, world_war_end_year
5. def main():
6. alliance, war_end_year = world_war()
7. print(f"The war was won by {alliance} and the war ended in {war_end_year}")
8. if __name__ == "__main__":
9. main()
```

Output

Which alliance won World War 2? Allies
When did World War 2 end? 1945
The war was won by Allies and the war ended in 1945

Program 4.5: Calculate the Value of sin(x) up to n Terms Using the Series
sin (*x*) = *x x* + *x x* + 1! 3! 5! 7! − − 3 5 7 _ Where x Is in Degrees

```
1. import math
2. degrees = int(input("Enter the degrees"))
3. nterms = int(input("Enter the number of terms"))
4. radians = degrees * math.pi / 180
5. def calculate_sin():
6. result = 0
7. numerator = radians
8. denominator = 1
9. for i in range(1, nterms+1):
10. single_term = numerator / denominator
11. result = result + single_term
```

```
12. numerator = -numerator * radians * radians
13. denominator = denominator * (2 * i) * (2 * i + 1)
14. return result
15. def main():
16. result = calculate_sin()
17. print(f"value is sin(x) calculated using the series is {result} ")
18. if __name__ == "__main__":
19. main()
```

Output

```
Enter the degrees45
Enter the number of terms5
value is sin(x) calculated using the series is 0.7071067829368671
```

Program to Check If a 3 Digit Number Is Armstrong Number or Not

```
1. user_number = int(input("Enter a 3 digit positive number to check for Armstrong
number"))
2. def check_armstrong_number(number):
3. result = 0
4. temp = number
5. while temp != 0:
6. last_digit = temp % 10
7. result += pow(last_digit, 3)
8. temp = int(temp / 10)
9. if number == result:
10. print(f"Entered number {number} is a Armstrong number")
11. else:
12. print(f"Entered number {number} is not a Armstrong number")
13. def main():
14. check_armstrong_number(user_number)
15. if __name__ == "__main__":
16. main()
```

Output

```
Enter a 3-digit positive number to check for Armstrong number407
Entered number 407 is a Armstrong number
```

Program to Demonstrate the Scope of Variables

```
1. test_variable = 5
2. def outer_function():
3. test_variable = 60
4. def inner_function():
5. test_variable = 100
6. print(f"Local variable value of {test_variable} having local scope to inner function
```

```
is displayed")
7. inner_function()
8. print(f"Local variable value of {test_variable} having local scope to outer function
is displayed ")
9. outer_function()
10. print(f"Global variable value of {test_variable} is displayed ")
```

Output

```
Local variable value of 100 having local scope to inner function is displayed
Local variable value of 60 having local scope to outer function is displayed
Global variable value of 5 is displayed
```

Calculate and Add the Surface Area of Two Cubes Use Nested Functions

```
1. def add_cubes(a, b):
2. def cube_surface_area(x):
3. return 6 * pow(x, 2)
4. return cube_surface_area(a) + cube_surface_area(b)
5. def main():
6. result = add_cubes(2, 3)
7. print(f"The surface area after adding two Cubes is {result}")
8. if __name__ == "__main__":
9. main()
```

Output

```
The surface area after adding two Cubes is 78
```

4.12 Packages

A package in Python is a directory that contains a special file called __init__.py and other modules or sub-packages. The __init__.py file can be empty or contain initialization code for the package. Packages allow for a hierarchical organization of modules, making it easier to manage and organize code.

Components of a Package

1. **Package Directory**: A directory that contains one or more Python modules or sub-packages. The directory name typically represents the package name.
2. **__init__.py**: A special Python file that initializes the package. This file is executed when the package is imported. It can be empty, but it often contains initialization code or package-level variables.
3. **Modules**: Python files (*.py) within the package directory that define functions, classes, and variables.
4. **Sub-packages**: Packages within a package, creating a hierarchical structure.

Example Package Structure

Here is an example of a simple package structure:

```
my_package/
    __init__.py
```

```
module1.py
module2.py
sub_package/
    __init__.py
    module3.py
```

- my_package is the top-level package directory.
- __init__.py is the package initializer.
- module1.py and module2.py are modules in the my_package package.
- sub_package is a sub-package of my_package.
- sub_package/__init__.py initializes the sub_package.
- sub_package/module3.py is a module within sub_package.

Creating a Package

To create a package, follow these steps:

1. **Create a Directory for the Package**

Create a directory with the desired package name. For example, my_package.

```
mkdir my_package
```

2. **Add an __init__.py File**

Create an empty __init__.py file inside the package directory. This file can also contain initialization code if needed.

```
touch my_package/__init__.py
```

3. **Add Modules**

Create Python files (modules) inside the package directory.

```
touch my_package/module1.py
touch my_package/module2.py
```

4. **(Optional) Create Sub-packages**

To create sub-packages, repeat the above steps inside a sub-directory.

```
mkdir my_package/sub_package
touch my_package/sub_package/__init__.py
touch my_package/sub_package/module3.py
```

Using a Package

To use a package in your code, you import it using Python's import statement.

Importing Modules from a Package

You can import modules from a package in several ways:

1. **Import an Entire Module**

```
import my_package.module1
```

2. **Import Specific Functions or Classes**

```
from my_package.module1 import my_function
```

3. **Import with an Alias**

```
import my_package.module1 as m1
```

4. **Import All Items from a Module**

```
from my_package.module1 import *
```

Example Usage

Assuming the following code in my_package/module1.py:

```
def my_function():
    return "Hello from module1!"
```

You can use it in another script as follows:

```
from my_package.module1 import my_function

print(my_function())  # Output: Hello from module1!
```

Package Initialization

The __init__.py file in a package can contain code that initializes the package when it is imported. This might include defining package-level variables, importing specific modules, or setting up logging.

Example: my_package/__init__.py

```
print("Initializing my_package")

# Import specific functions or modules for convenience
from .module1 import my_function
from .module2 import another_function

# Define package-level variables
__version__ = '1.0.0'
```

Accessing Package-Level Variables

You can access package-level variables directly after importing the package:

```
import my_package

print(my_package.__version__)  # Output: 1.0.0
```

Relative and Absolute Imports

Absolute Imports

Absolute imports specify the full path from the root of the package hierarchy:

```
from my_package.module1 import my_function
from my_package.sub_package.module3 import another_function
```

Relative Imports

Relative imports specify the path relative to the current module's position:

```
from .module1 import my_function  # Same package
from .sub_package.module3 import another_function  # Sub-package
```

Note: Relative imports work within packages and are typically used in modules within the same package or sub-packages.

Summary

- **Packages**: Directories containing modules and an __init__.py file, allowing for hierarchical organization of related code.
- **__init__.py**: Initializes the package and can contain initialization code or package-level variables.
- **Creating Packages**: Create a directory, add an __init__.py file, and include modules and sub-packages as needed.
- **Using Packages**: Import modules or functions using absolute or relative imports.
- **Initialization**: Use __init__.py for package initialization and to define package-level variables.

UNIT 5

Python Object Oriented

Topic No.	Topic
5.1	Overview of OOP
5.2	Creating Classes and Objects
5.3	Accessing attributes
5.4	Built-In Class Attributes
5.5	Destroying Objects

5.1 Overview of OOP

1. **Class**:
    - A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects (instances) will have.
    - Classes encapsulate data and behavior.
    - **Syntax**: In Python, a class is defined using the class keyword.

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

```
        self.model = model
        self.year = year

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")
```

2. **Object**:
    - An object is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created.
    - Objects have attributes (data) and methods (functions) that define their behavior.

```
my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info()  # Output: 2020 Toyota Corolla
```

3. **Attributes**:
    - Attributes are variables that belong to an object or class. They store data about the object.
    - There are instance attributes and class attributes. Instance attributes are unique to each instance, while class attributes are shared among all instances of the class.

```
class Car:
    wheels = 4  # Class attribute

    def __init__(self, make, model, year):
        self.make = make  # Instance attribute
        self.model = model
        self.year = year
```

4. **Methods**:
    - Methods are functions defined within a class that describe the behaviors of the objects.
    - Instance methods, class methods, and static methods are the types of methods.

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

    def display_info(self):  # Instance method
        print(f"{self.year} {self.make} {self.model}")

    @classmethod
    def is_motor_vehicle(cls):  # Class method
        return True

    @staticmethod
```

```
    def honk():  # Static method
        print("Honk!")
```

5. **Encapsulation**:
    - Encapsulation is the concept of bundling data and methods that operate on the data within one unit, usually a class. It restricts direct access to some of the object's components and can prevent the accidental modification of data.
    - Attributes are often marked as private using an underscore prefix (_) to indicate they should not be accessed directly.

```
class Car:
    def __init__(self, make, model, year):
        self._make = make  # Private attribute
        self._model = model
        self._year = year

    def display_info(self):
        print(f"{self._year} {self._make} {self._model}")

    def set_year(self, year):
        if year > 1885:  # The first car was invented around 1885
            self._year = year
        else:
            print("Invalid year")
```

6. **Inheritance**:
    - Inheritance is a way to form new classes using classes that have already been defined. It allows for a hierarchy of classes that share a set of attributes and methods.
    - The new class (derived class) inherits the attributes and methods of the existing class (base class).

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

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")

class Car(Vehicle):
    def __init__(self, make, model, year, fuel_type):
        super().__init__(make, model, year)  # Call the base class constructor
        self.fuel_type = fuel_type

    def display_info(self):
        super().display_info()
        print(f"Fuel type: {self.fuel_type}")
```

7. **Polymorphism**:
    - Polymorphism means "many shapes" and allows methods to do different things based on the object it is acting upon, even if they share the same name.
    - In Python, polymorphism can be implemented through method overriding and operator overloading.

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

def make_animal_speak(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()
make_animal_speak(dog)  # Output: Woof!
make_animal_speak(cat)  # Output: Meow!
```

8. **Abstraction**:
    - Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object.
    - Abstract classes and interfaces can be used to define a set of methods that must be created within any child classes built from the abstract class.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

Benefits of OOP

1. **Modularity**: Code is organized into classes and objects, making it modular and easier to manage.
2. **Reusability**: Classes and objects can be reused across different programs or parts of a program.
3. **Scalability**: OOP allows for the easy addition of new features and functionality.
4. **Maintainability**: Easier to maintain and update code due to its modular nature.
5. **Encapsulation**: Helps protect data and ensures that only relevant parts of the code can access or modify data.

Summary

- **Classes and Objects**: Classes are blueprints for objects, encapsulating data and methods.
- **Encapsulation**: Bundles data and methods, restricting direct access to some components.
- **Inheritance**: Allows new classes to inherit attributes and methods from existing classes.
- **Polymorphism**: Enables methods to operate differently based on the object they are called on.
- **Abstraction**: Hides complex implementation details, showing only essential features.

OOP is a powerful paradigm that supports the creation of robust, scalable, and maintainable code. By leveraging these concepts, developers can build complex software systems in a more manageable and efficient way.

5.2 Creating Classes and Objects

Creating Classes

A class in Python is defined using the class keyword. Classes serve as blueprints for creating objects and encapsulate data and behavior in the form of attributes and methods.

Basic Structure of a Class

```
class ClassName:
    # Class attribute
    class_attribute = "This is a class attribute"

    def __init__(self, instance_attribute):
        # Instance attribute
        self.instance_attribute = instance_attribute

    # Method
    def method_name(self):
        print("This is a method")
```

Components of a Class

1. **Class Name**: The name of the class, which should follow standard naming conventions (PascalCase).
2. **Class Attribute**: A variable that is shared among all instances of the class.
3. **Instance Attribute**: A variable that is unique to each instance of the class.
4. **Methods**: Functions defined within the class that describe the behavior of the objects.

Example Class Definition

```
class Car:
    # Class attribute
    wheels = 4

    def __init__(self, make, model, year):
        # Instance attributes
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        # Method
        print(f"{self.year} {self.make} {self.model}")
```

Creating Objects
An object is an instance of a class. When a class is defined, no memory is allocated until an object of that class is instantiated.

Creating an Object
To create an object, you call the class using its name followed by parentheses, optionally passing any arguments required by the __init__ method.

```
my_car = Car("Toyota", "Corolla", 2020)
```

Accessing Attributes and Methods
Once an object is created, you can access its attributes and methods using dot notation.

```
print(my_car.make)  # Output: Toyota
print(my_car.model)  # Output: Corolla
print(my_car.year)  # Output: 2020

my_car.display_info()  # Output: 2020 Toyota Corolla
```

Example Usage

```
class Car:
    wheels = 4

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

```
        self.year = year

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes
print(my_car.make)  # Output: Toyota
print(my_car.model)  # Output: Corolla
print(my_car.year)  # Output: 2020

# Calling a method
my_car.display_info()  # Output: 2020 Toyota Corolla
```

Class and Instance Attributes

Class Attributes
Class attributes are shared among all instances of the class. They are defined within the class but outside any methods.

```
class Car:
    wheels = 4  # Class attribute

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

# Accessing class attribute
print(Car.wheels)  # Output: 4
```

Instance Attributes
Instance attributes are unique to each instance. They are defined within the __init__ method or other methods and are accessed using self.

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

# Creating objects
car1 = Car("Toyota", "Corolla", 2020)
car2 = Car("Honda", "Civic", 2021)

# Accessing instance attributes
print(car1.make)  # Output: Toyota
print(car2.make)  # Output: Honda
```

Methods in Classes

Instance Methods
Instance methods are functions defined within a class that operate on instances of the class. They must have self as their first parameter.

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

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")

# Calling an instance method
my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info()  # Output: 2020 Toyota Corolla
```

Class Methods
Class methods are methods that are bound to the class and not the instance. They can modify class state that applies across all instances of the class. They must have cls as their first parameter and are defined using the @classmethod decorator.

```
class Car:
    wheels = 4

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

    @classmethod
    def change_wheels(cls, new_wheels):
        cls.wheels = new_wheels

# Calling a class method
Car.change_wheels(6)
print(Car.wheels)  # Output: 6
```

Static Methods
Static methods do not modify class or instance state. They are defined using the @staticmethod decorator.

```
class Car:
    @staticmethod
    def is_motor_vehicle():
```

```
        return True

# Calling a static method
print(Car.is_motor_vehicle())  # Output: True
```

Encapsulation

Encapsulation is the concept of restricting access to certain components of an object and can prevent the accidental modification of data. It is implemented by using private attributes and methods (by convention, these are prefixed with an underscore _).

```
class Car:
    def __init__(self, make, model, year):
        self._make = make  # Private attribute
        self._model = model
        self._year = year

    def display_info(self):
        print(f"{self._year} {self._make} {self._model}")

    def set_year(self, year):
        if year > 1885:
            self._year = year
        else:
            print("Invalid year")

# Accessing private attributes
my_car = Car("Toyota", "Corolla", 2020)
print(my_car._make)  # Output: Toyota (though accessible, it should be avoided)
my_car.set_year(1880)  # Output: Invalid year
```

Inheritance

Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class). This promotes code reusability and creates a hierarchical relationship between classes.

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

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")

class Car(Vehicle):
    def __init__(self, make, model, year, fuel_type):
        super().__init__(make, model, year)
        self.fuel_type = fuel_type

    def display_info(self):
```

```
        super().display_info()
        print(f"Fuel type: {self.fuel_type}")

# Creating an object of the child class
my_car = Car("Toyota", "Corolla", 2020, "Gasoline")
my_car.display_info()
# Output:
# 2020 Toyota Corolla
# Fuel type: Gasoline
```

Polymorphism
Polymorphism allows methods to do different things based on the object they are acting upon. This can be achieved through method overriding and operator overloading.

Method Overriding
When a method in a child class has the same name as a method in the parent class, the child class method overrides the parent class method.

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Demonstrating polymorphism
animals = [Dog(), Cat()]

for animal in animals:
    print(animal.speak())
# Output:
# Woof!
# Meow!
```

Summary

- **Classes and Objects**: Classes are blueprints for objects. Objects are instances of classes.
- **Attributes**: Class attributes are shared among all instances. Instance attributes are unique to each instance.
- **Methods**: Instance methods operate on instances, class methods operate on the class, and static methods do not modify class or instance state.
- **Encapsulation**: Restricts access to certain components to prevent accidental modification.

- **Inheritance**: Child classes inherit attributes and methods from parent classes.
- **Polymorphism**: Methods can do different things based on the object they act upon.

5.3 Accessing attributes

Accessing Attributes

1. Class Attributes
Class attributes are defined within a class but outside any methods. They are shared among all instances of the class. To access a class attribute, you can use the class name or an instance of the class.

Defining and Accessing Class Attributes

```
class Car:
    wheels = 4  # Class attribute

# Accessing class attribute using the class name
print(Car.wheels)  # Output: 4

# Accessing class attribute using an instance
my_car = Car()
print(my_car.wheels)  # Output: 4
```

2. Instance Attributes
Instance attributes are defined within methods (usually the __init__ method) and are unique to each instance of the class. To access an instance attribute, you use the instance name followed by the attribute name.

Defining and Accessing Instance Attributes

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

# Creating an instance of the Car class
my_car = Car("Toyota", "Corolla", 2020)

# Accessing instance attributes
print(my_car.make)  # Output: Toyota
print(my_car.model)  # Output: Corolla
print(my_car.year)  # Output: 2020
```

3. Modifying Attributes

Modifying Class Attributes

Class attributes can be modified using the class name or an instance of the class. Changing a class attribute using the class name affects all instances, while changing it using an instance only affects that specific instance.

```
class Car:
    wheels = 4

# Modifying class attribute using the class name
Car.wheels = 6
print(Car.wheels)  # Output: 6

# Modifying class attribute using an instance
my_car = Car()
my_car.wheels = 8
print(my_car.wheels)  # Output: 8
print(Car.wheels)  # Output: 6
```

Modifying Instance Attributes

Instance attributes can be modified directly using the instance name followed by the attribute name.

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

# Creating an instance of the Car class
my_car = Car("Toyota", "Corolla", 2020)

# Modifying instance attributes
my_car.make = "Honda"
my_car.model = "Civic"
my_car.year = 2021

# Accessing modified instance attributes
print(my_car.make)  # Output: Honda
print(my_car.model)  # Output: Civic
print(my_car.year)  # Output: 2021
```

4. Deleting Attributes

Attributes can be deleted using the del keyword followed by the instance name and the attribute name.

Deleting Instance Attributes

```
class Car:
```

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

# Creating an instance of the Car class
my_car = Car("Toyota", "Corolla", 2020)

# Deleting an instance attribute
del my_car.year

# Trying to access the deleted attribute will raise an AttributeError
try:
    print(my_car.year)
except AttributeError as e:
    print(e)  # Output: 'Car' object has no attribute 'year'
```

Deleting Class Attributes

```
class Car:
    wheels = 4

# Deleting a class attribute
del Car.wheels

# Trying to access the deleted attribute will raise an AttributeError
try:
    print(Car.wheels)
except AttributeError as e:
    print(e)  # Output: type object 'Car' has no attribute 'wheels'
```

5. Private and Protected Attributes

Protected Attributes
By convention, protected attributes are prefixed with a single underscore _. They are meant to be accessed only within the class and its subclasses.

```
class Car:
    def __init__(self, make, model, year):
        self._make = make  # Protected attribute
        self._model = model
        self._year = year

# Accessing protected attributes
my_car = Car("Toyota", "Corolla", 2020)
print(my_car._make)  # Output: Toyota
```

Private Attributes
By convention, private attributes are prefixed with a double underscore __. They are intended to be accessed only within the class itself.

```
class Car:
    def __init__(self, make, model, year):
        self.__make = make  # Private attribute
        self.__model = model
        self.__year = year

    def get_make(self):
        return self.__make

# Accessing private attributes (not recommended)
my_car = Car("Toyota", "Corolla", 2020)
try:
    print(my_car.__make)
except AttributeError as e:
    print(e)  # Output: 'Car' object has no attribute '__make'

# Accessing private attributes via a public method
print(my_car.get_make())  # Output: Toyota
```

6. Using Property Decorators

Property decorators provide a way to manage access to instance attributes. They allow you to define methods that get, set, and delete instance attributes.

Example with Property Decorators

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

    @property
    def make(self):
        return self._make

    @make.setter
    def make(self, value):
        self._make = value

    @make.deleter
    def make(self):
        del self._make

# Using property decorators
my_car = Car("Toyota", "Corolla", 2020)

# Getting the value
print(my_car.make)  # Output: Toyota
```

```
# Setting the value
my_car.make = "Honda"
print(my_car.make)  # Output: Honda

# Deleting the value
del my_car.make
try:
    print(my_car.make)
except AttributeError as e:
    print(e)  # Output: 'Car' object has no attribute '_make'
```

Summary

- **Class Attributes**: Shared among all instances, accessed via class name or instance.
- **Instance Attributes**: Unique to each instance, accessed via instance.
- **Modifying Attributes**: Can be done directly using the class or instance name followed by the attribute name.
- **Deleting Attributes**: Using the del keyword.
- **Private and Protected Attributes**: Use single underscore _ for protected and double underscore __ for private attributes.
- **Property Decorators**: Allow controlled access to instance attributes through methods.

This comprehensive guide covers the essential aspects of accessing and managing attributes in Python classes

5.4 Built-In Class Attributes

Common Built-in Class Attributes

1. __dict__

The __dict__ attribute is a dictionary that stores the class's namespace. It contains all the attributes and methods defined for the class.

Example:

```
class MyClass:
    class_var = 42

    def __init__(self, instance_var):
        self.instance_var = instance_var

    def method(self):
        pass

# Accessing __dict__ attribute
print(MyClass.__dict__)
```

Output:

{'__module__': '__main__', 'class_var': 42, '__init__': <function MyClass.__init__ at 0x7f1b3c5a7a60>, 'method': <function MyClass.method at 0x7f1b3c5a7af0>, '__dict__': <attribute '__dict__' of 'MyClass' objects>, '__weakref__': <attribute '__weakref__' of 'MyClass' objects>, '__doc__': None}

2. __doc__

The __doc__ attribute contains the docstring of the class, which is a string literal that occurs as the first statement in a class definition. This is used for documentation purposes.

Example:

```
class MyClass:
    """
    This is a docstring for MyClass.
    """
    pass

# Accessing __doc__ attribute
print(MyClass.__doc__)
```

Output:

This is a docstring for MyClass.

3. __name__
The __name__ attribute contains the name of the class.

Example:

```
class MyClass:
    pass

# Accessing __name__ attribute
print(MyClass.__name__)
```

Output:
MyClass

4. __module__
The __module__ attribute contains the name of the module in which the class was defined.

Example:

```
class MyClass:
    pass
# Accessing __module__ attribute
print(MyClass.__module__)
```

Output:

__main__

5. __bases__
The __bases__ attribute is a tuple containing the base classes of the class. If the class does not inherit from any other class, this will be an empty tuple.

Example:
```
class BaseClass:
    pass
class DerivedClass(BaseClass):
    pass
# Accessing __bases__ attribute
print(DerivedClass.__bases__)
```

Output:
(<class '__main__.BaseClass'>,)

6. __mro__
The __mro__ (Method Resolution Order) attribute is a tuple that shows the order in which base classes are searched when looking for a method. This is particularly useful in the context of multiple inheritance.

Example:
```
class BaseClass1:
    pass
class BaseClass2:
    pass
class DerivedClass(BaseClass1, BaseClass2):
    pass
# Accessing __mro__ attribute
print(DerivedClass.__mro__)
```

Output:
(<class '__main__.DerivedClass'>, <class '__main__.BaseClass1'>, <class '__main__.BaseClass2'>, <class 'object'>)

7. __class__
The __class__ attribute returns the class of an instance.

Example:
```
class MyClass:
    pass
# Creating an instance of MyClass
instance = MyClass()
# Accessing __class__ attribute
print(instance.__class__)
```

Output:
<class '__main__.MyClass'>

8. __qualname__
The __qualname__ attribute contains the qualified name of the class. This includes the name of the class and any enclosing classes or functions.

Example:
```
class OuterClass:
    class InnerClass:
        pass
# Accessing __qualname__ attribute
print(OuterClass.InnerClass.__qualname__)
```

Output:
OuterClass.InnerClass

Summary
Built-in class attributes in Python provide essential information about the class, such as its name, documentation, module, base classes, and more. These attributes can be extremely useful for debugging, documentation, and understanding the structure of your classes.

Common Built-in Class Attributes:
1. **__dict__**: Dictionary containing the class's namespace.
2. **__doc__**: Docstring of the class.
3. **__name__**: Name of the class.
4. **__module__**: Name of the module in which the class was defined.
5. **__bases__**: Tuple containing the base classes of the class.
6. **__mro__**: Tuple showing the method resolution order.
7. **__class__**: Class of an instance.
8. **__qualname__**: Qualified name of the class.

5.5 Destroying Objects

Concepts of Destroying Objects

1. Garbage Collection
Python uses an automatic garbage collection system to manage memory. This system tracks objects and their references and automatically frees memory when objects are no longer in use. The primary mechanism for this is reference counting, supplemented by a cyclic garbage collector to handle reference cycles.

Reference Counting
Each object in Python maintains a count of references to it. When the reference count drops to zero, the object is deallocated.

Example:
```
class MyClass:
    def __init__(self, value):
        self.value = value
obj = MyClass(10)
```

```
print(obj.value)  # Output: 10
del obj  # Decrease reference count to 0, object is destroyed
```

2. __del__ Method (Destructor)

The __del__ method, also known as the destructor, is called when an object is about to be destroyed. This method can be overridden to define custom cleanup behavior.

Example:

```
class MyClass:
    def __init__(self, value):
        self.value = value
        print(f"Object with value {value} is created.")

    def __del__(self):
        print(f"Object with value {self.value} is destroyed.")

obj = MyClass(10)
del obj  # Explicitly delete the object
```

Output:

```
Object with value 10 is created.
Object with value 10 is destroyed.
```

3. Explicitly Deleting Objects

Objects can be explicitly deleted using the del statement, which reduces the reference count of the object by one. If the reference count drops to zero, the object is destroyed.

Example:

```
class MyClass:
    def __init__(self, value):
        self.value = value

obj1 = MyClass(10)
obj2 = obj1  # Increase reference count

del obj1  # Decrease reference count but obj2 still references the object
del obj2  # Now the reference count is zero, object is destroyed
```

4. Weak References

Weak references allow an object to be referenced without increasing its reference count. This is useful for caching and other scenarios where you want to hold a reference to an object without preventing its destruction.

Example:

```
import weakref
class MyClass:
    def __init__(self, value):
```

```
        self.value = value
obj = MyClass(10)
weak_obj = weakref.ref(obj)  # Create a weak reference
print(weak_obj().value)  # Output: 10
del obj  # The original object is deleted
print(weak_obj())  # Output: None (the weak reference returns None)
```

5. Circular References

Circular references occur when two or more objects reference each other, creating a cycle. This can prevent the reference count from dropping to zero. Python's cyclic garbage collector can detect and collect these cycles.

Example:

```
class A:
    def __init__(self):
        self.b = None
class B:
    def __init__(self):
        self.a = None
a = A()
b = B()
a.b = b
b.a = a
del a
del b  # Even though a and b reference each other, the garbage collector can handle this
```

6. Manual Garbage Collection

Python provides the gc module to interact with the garbage collector, allowing you to manually trigger garbage collection or adjust its behavior.

Example:

```
import gc
# Disable automatic garbage collection
gc.disable()
# Enable garbage collection
gc.enable()
# Manually trigger garbage collection
gc.collect()
```

Summary

- **Garbage Collection**: Python's automatic memory management system that uses reference counting and cyclic garbage collection.
- **__del__ Method**: A destructor method that can be overridden for custom cleanup behavior.
- **Explicit Deletion**: Use the del statement to explicitly delete objects and manage reference counts.
- **Weak References**: Allow referencing objects without increasing reference counts, useful for caching.

- **Circular References**: Cycles in references handled by Python's cyclic garbage collector.
- **Manual Garbage Collection**: Use the gc module to manually control garbage collection.

Given Three Points (x1, y1), (x2, y2) and (x3, y3), Write a Python Program to Check If they are Collinear

```
1. class Collinear:
2. def __init__(self, x, y):
3. self.x_coord = x
4. self.y_coord = y
5. def check_for_collinear(self, point_2_obj, point_3_obj):
6. if (point_3_obj.y_coord - point_2_obj.y_coord)*(point_2_obj.x_coord -
self.x_coord) == (point_2_obj.y_coord - self.y_coord)*(point_3_obj.x_coord -
point_2_obj.x_coord):
7. print("Points are Collinear")
8. else:
9. print("Points are not Collinear")
10. def main():
11. point_1 = Collinear(1, 5)
12. point_2 = Collinear(2, 5)
13. point_3 = Collinear(4, 6)
14. point_1.check_for_collinear(point_2, point_3)
15. if __name__ == "__main__":
16. main()
```

Program to Demonstrate the Difference between Abstraction and Encapsulation

```
1. class foo:
2. def __init__(self, a, b):
3. self.a = a
4. self.b = b
5. def add(self):
6. return self.a + self.b
7. foo_object = foo(3,4)
8. foo_object.add()
```

Given a point(x, y), Write Python Program to Find Whether it Lies in the First, Second, Third or Fourth Quadrant of x - y Plane

```
1. class Quadrant:
2. def __init__(self, x, y):
3. self.x_coord = x
4. self.y_coord = y
5. def determine_quadrant(self):
6. if self.x_coord > 0 and self.y_coord > 0:
7. print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the FIRST
Quadrant")
8. elif self.x_coord < 0 and self.y_coord < 0:
```

```
9. print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the
THIRD Quadrant")
10. elif self.x_coord > 0 and self.y_coord < 0:
11. print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the
FOURTH Quadrant")
12. elif self.x_coord < 0 and self.y_coord > 0:
13. print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the
SECOND Quadrant")
14. def main():
15. point = Quadrant(-180, 180)
16. point.determine_quadrant()
17. if __name__ == "__main__":
18. main()
```

Program to Demonstrate Multiple Inheritance with Method Overriding

```
1. class Pet:
2. def __init__(self, breed):
3. self.breed = breed
4. def about(self):
5. print(f"This is {self.breed} breed")
6. class Insurable:
7. def __init__(self, amount):
8. self.amount = amount
9. def about(self):
10. print(f"Its insured for an amount of {self.amount}")
11. class Cat(Pet, Insurable):
12. def __init__(self, weight, breed, amount):
13. self.weight = weight
14. Pet.__init__(self, breed)
15. Insurable.__init__(self, amount)
16. def get_weight(self):
17. print(f"{self.breed} Cat weighs around {self.weight} pounds")
18. def main():
19. cat_obj = Cat(15, "Ragdoll", "$100")
20. cat_obj.about()
21. cat_obj.get_weight()
22. if __name__ == "__main__":
23. main()
```

UNIT 6
Python Exceptions Handling

Topic No.	Topic
6.1	What is Exception
6.2	Handling an exception: try except else
6.3	Try finally clause
6.4	Arguments in Exception
6.5	Python Standard Exception Raising and exceptions
6.6	User Defined Exception

6.1 What is Exception?

Key Concepts of Exceptions in Python

1. **Definition of an Exception**: An exception is an object that represents an error or an unusual condition that a program encounters. When an exception is raised, it signifies that something unexpected has happened which requires special handling.

2. **Types of Exceptions**: Python has a wide range of built-in exceptions that cover common error conditions. These include:
    - **SyntaxError**: Raised when the code has incorrect syntax.
    - **TypeError**: Raised when an operation is applied to an object of inappropriate type.
    - **ValueError**: Raised when a function receives an argument of the right type but inappropriate value.
    - **IndexError**: Raised when trying to access an element from a list using an index that is out of range.
    - **KeyError**: Raised when trying to access a dictionary with a key that does not exist.
    - **FileNotFoundError**: Raised when trying to open a file that does not exist.

3. **Exception Hierarchy**: Exceptions in Python are organized in a hierarchy. The base class for all built-in exceptions is BaseException. All other exceptions are derived from this class, either directly or indirectly. The most common base class for user-defined exceptions is Exception.

Example hierarchy:

- BaseException
    - Exception
        - ArithmeticError
        - LookupError
            - IndexError
            - KeyError
        - ValueError
        - TypeError

4. **Creating Custom Exceptions**: Python allows the creation of user-defined exceptions by subclassing the built-in Exception class or any of its subclasses. This enables developers to define their own error types that are relevant to their specific needs.

Example:

```
class MyCustomError(Exception):
    """Custom exception for specific errors."""
    pass
```

5. **Exception Propagation**: When an exception occurs, it propagates up the call stack. This means that if a function raises an exception, and it is not handled within that function, the exception is passed to the calling function. This continues until the exception is either caught and handled or reaches the top level of the program.

6. **Exception Objects**: Exceptions are objects that can hold information about the error, such as an error message. These objects are instances of exception classes.

Example:

```
try:
    x = int("not a number")
except ValueError as e:
    print(e)  # Output: invalid literal for int() with base 10: 'not a number'
```

Summary

- **Exception**: An event that disrupts the normal execution flow of a program, representing errors or unusual conditions.
- **Types of Exceptions**: Python has built-in exceptions for various error conditions (e.g., SyntaxError, TypeError, IndexError).
- **Exception Hierarchy**: Exceptions are organized in a hierarchy, with BaseException as the root.
- **Custom Exceptions**: Users can create custom exceptions by subclassing the Exception class.
- **Exception Propagation**: Exceptions propagate up the call stack until handled or reaching the program's top level.
- **Exception Objects**: Instances of exception classes that hold information about the error.

6.2 Handling an exception try….except…else

1. try Block

The try block is where you place code that you suspect might raise an exception. It's used to wrap potentially problematic code so that you can handle any issues that arise in a controlled way.

- **Purpose**: To execute code that might fail or generate exceptions.
- **Typical Usage**: Commonly used for operations such as file I/O, network requests, or mathematical operations that could result in errors.

Example:

```
try:
    result = 10 / 2  # This will execute without an issue
    print("Result is", result)
```

Here, the try block attempts to divide 10 by 2, which is successful, so the code proceeds without raising an exception.

2. except Block

The except block follows the try block and handles exceptions that occur within the try block. You can specify different types of exceptions to handle various errors in distinct ways.

- **Purpose**: To catch and handle exceptions that occur in the try block.
- **Multiple except Blocks**: You can have multiple except blocks to handle different types of exceptions separately.

Example:

```
try:
    result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero.")  # Handling ZeroDivisionError
```

In this example, dividing by zero raises a ZeroDivisionError, which is then caught and handled by the except block.

Handling Multiple Exceptions:
You can catch multiple exceptions using multiple except blocks:

```
try:
    value = int(input("Enter a number: "))
    result = 10 / value
except ValueError:
    print("Invalid input. Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Here, two exceptions are handled: ValueError for invalid input and ZeroDivisionError for division by zero.

Catching All Exceptions:
You can catch all exceptions by using a generic except block. However, this is not recommended unless necessary, as it can hide unexpected issues.

```
try:
    result = 10 / 0
except:
    print("An unexpected error occurred.")
```

3. else Block
The else block is executed if the try block completes successfully without raising any exceptions. This block is useful for code that should only run if no errors occurred.

- **Purpose**: To execute code that should only run if no exceptions were raised.
- **Placement**: It must be placed after all except blocks.

Example:

```
try:
    result = 10 / 2  # No exception is raised
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("The result is", result)  # This runs because no exception was raised
```

In this example, since there's no exception raised, the else block executes, printing the result.

6.3 Try-finally clause

try and finally Blocks

In Python, the try block is used to wrap code that might raise an exception, while the finally block is used to ensure that certain code is executed no matter what happens in the try block. This is particularly useful for resource management and cleanup tasks that must be completed regardless of whether an exception occurs.

1. try Block

- **Purpose**: The try block is used to enclose code that you want to test for exceptions. It contains code that might potentially raise an exception.
- **Behavior**: Python attempts to execute the code inside the try block. If an exception occurs, Python stops executing the rest of the try block and jumps to the corresponding except block if present, or continues to the finally block if no except block is provided.

Example:

```
try:
   result = 10 / 2  # This will execute without issues
   print("Result is", result)
```

In this example, the try block runs without any issues, so no exception is raised.

2. finally Block

- **Purpose**: The finally block is used to define cleanup actions that must be executed under all circumstances, regardless of whether an exception was raised in the try block.
- **Behavior**: The code in the finally block will always be executed after the try block, even if an exception is raised and handled or if no exception occurs at all.

Example:

```
try:
   file = open("example.txt", "r")
   data = file.read()
finally:
   file.close()  # This will always be executed
   print("File closed.")
```

In this example:

- **try Block**: Attempts to open and read from a file.
- **finally Block**: Ensures that the file is closed regardless of whether an exception occurs (e.g., file not found, read error).
-

Interaction Between try and finally

1. **Normal Execution**:
    - If the try block executes without any exceptions, the finally block will execute after the try block completes.
2. **Exception Handling**:
    - If an exception is raised in the try block, and there is no except block, the finally block will still execute before the exception is propagated further.
    - If there is an except block, it will handle the exception, and then the finally block will execute.
    -

Example with Exception:

```
try:
    print("Trying to open file...")
    file = open("example.txt", "r")
    data = file.read()
    raise ValueError("An error occurred!")  # Simulating an error
finally:
    print("Closing file...")
    file.close()  # Ensures the file is closed even if an error occurs
```

In this example:

- **try Block**: Attempts to open and read a file, then raises a ValueError.
- **finally Block**: The file is closed regardless of the error.

Practical Uses

1. **Resource Management**:
    - Ensuring resources like files, network connections, or database connections are properly closed or released after use.
2. **Cleanup Actions**:
    - Performing necessary cleanup actions such as releasing locks, cleaning up temporary files, or resetting system states.
3. **Unconditional Execution**:
    - Code in the finally block will always run, making it ideal for actions that should always be executed, such as logging or cleanup tasks.

Summary

- **try Block**: Contains code that might raise exceptions. Python attempts to execute this code and handles exceptions if they occur.
- **finally Block**: Contains code that must run regardless of whether an exception was raised or not. Ensures that cleanup actions are performed.

Example with Complete Structure

Here is a complete example demonstrating the usage of try, finally, and handling an exception:

```
try:
    # Code that might raise an exception
    file = open("example.txt", "r")
    data = file.read()
    # Simulate an error
    result = 10 / 0
except ZeroDivisionError:
    print("Caught a division by zero error.")
finally:
    # Cleanup actions
    file.close()
    print("File closed.")

print("Program continues...")
```

In this example:

- **try Block**: Tries to open a file and perform a division operation.
- **except Block**: Catches the ZeroDivisionError.
- **finally Block**: Ensures the file is closed regardless of the error.

6.4 Arguments in Exception.

Arguments in Exceptions

In Python, exceptions can be raised with arguments that provide additional information about the error. These arguments are used to give more context or detail about what went wrong. When you create or raise an exception, you can pass arguments to it which can then be accessed when the exception is handled.

Here's a detailed explanation:

1. Raising Exceptions with Arguments

When raising exceptions, you can provide arguments to pass information about the error. These arguments are passed to the exception class's constructor.

Example:

raise ValueError("Invalid value provided")

In this example, "Invalid value provided" is the argument passed to the ValueError exception. This message can provide more details about what caused the exception.

2. Custom Exceptions with Arguments

You can also create custom exceptions that accept arguments. This allows you to define exceptions that carry specific information related to your application.

Defining Custom Exceptions:

To create a custom exception, subclass the built-in Exception class and define an __init__ method to accept and store arguments.

```
class MyCustomError(Exception):
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code
```

Raising Custom Exceptions:

You can then raise this custom exception with arguments:

raise MyCustomError("Something went wrong", 404)

3. Handling Exceptions with Arguments

When handling exceptions, you can access the arguments provided when the exception was raised. If you pass a message or other data, you can retrieve this information in the except block.

Example:

```
try:
    raise ValueError("A specific error message")
except ValueError as e:
    print(f"Exception message: {e}")
```
In this example:

- **Exception Raised**: ValueError is raised with the message "A specific error message".
- **Exception Handled**: The except block captures the exception as e, and e contains the message "A specific error message", which can be accessed and printed.

Custom Exception Handling:

```
class MyCustomError(Exception):
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code

try:
    raise MyCustomError("An error occurred", 123)
except MyCustomError as e:
    print(f"Error message: {e}")
    print(f"Error code: {e.code}")
```

In this example:

- **Exception Raised**: MyCustomError is raised with a message and a code.
- **Exception Handled**: The except block captures MyCustomError as e, and both the message and the code can be accessed.

Summary

- **Arguments in Exceptions**: When raising exceptions, you can provide arguments that give more context about the error.
- **Custom Exceptions**: You can create custom exceptions with specific arguments and store additional information.
- **Handling Exceptions**: Access the arguments passed to exceptions in the except block to get detailed information about the error.

6.5 Python Standard Exceptions Raising and exceptions

1. Python Standard Exceptions

Python has a rich set of built-in exceptions that are derived from the base Exception class. Each exception is designed to handle a specific type of error. Here are some of the most commonly used standard exceptions:

Common Standard Exceptions:

- **Exception**: The base class for all built-in exceptions. Most user-defined exceptions should inherit from this class.
- **ValueError**: Raised when a function receives an argument of the correct type but inappropriate value. For example, converting a string to an integer where the string is not a valid integer.

```
int("hello")  # Raises ValueError
```

- **TypeError**: Raised when an operation or function is applied to an object of inappropriate type. For instance, adding a string and an integer.

```
"string" + 10  # Raises TypeError
```

- **IndexError**: Raised when a sequence subscript is out of range. For example, accessing an index that does not exist in a list.

```
my_list = [1, 2, 3]
my_list[5]  # Raises IndexError
```

- **KeyError**: Raised when a dictionary key is not found. For instance, accessing a non-existent key in a dictionary.

```
my_dict = {'a': 1}
my_dict['b']  # Raises KeyError
```

- **FileNotFoundError**: Raised when a file operation (like open()) fails because the file does not exist.

```
open('nonexistent_file.txt')  # Raises FileNotFoundError
```

- **ZeroDivisionError**: Raised when a division operation is performed with zero as the divisor.

```
10 / 0  # Raises ZeroDivisionError
```

- **IOError**: Raised when an I/O operation (like reading or writing to a file) fails. This is a more general I/O error that can include issues like file not found, permission errors, etc.

```
with open('file.txt', 'w') as f:
    f.write("Hello")  # Might raise IOError if there are issues with the file system
```

- **ImportError**: Raised when an import statement fails to find the module or name specified.

```
import non_existent_module  # Raises ImportError
```

2. Raising Exceptions

Raising exceptions is a way to signal that an error or an exceptional situation has occurred. You can use the raise keyword to raise built-in or custom exceptions.

Syntax:

```
raise ExceptionType("Optional error message")
```

- **ExceptionType**: The type of the exception you want to raise (e.g., ValueError, TypeError).
- **"Optional error message"**: An optional string that describes the error. This message can be accessed later when handling the exception.

Example:

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

```
    if b == 0:
        raise ZeroDivisionError("You cannot divide by zero")
    return a / b

try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

In this example:

- The divide function raises a ZeroDivisionError with a message if the divisor b is zero.
- The except block catches the ZeroDivisionError and prints the error message.

3. Handling Exceptions

Handling exceptions allows you to respond to errors gracefully and prevent your program from crashing unexpectedly. The try and except blocks are used for this purpose.

Basic Structure:

```
try:
    # Code that might raise an exception
    pass
except ExceptionType:
    # Code to handle the exception
    Pass
```

Detailed Example:

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:
    print("Invalid input! Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("The result is", result)
finally:
    print("This code runs no matter what.")
```

In this example:

- **try Block**: Contains code that might raise exceptions.
- **except ValueError**: Handles errors related to invalid input.
- **except ZeroDivisionError**: Handles division by zero errors.
- **else Block**: Runs if no exceptions are raised.
- **finally Block**: Always runs, whether an exception occurred or not.

Summary

- **Standard Exceptions**: Python has built-in exceptions for common error situations, such as ValueError, TypeError, IndexError, etc.
- **Raising Exceptions**: Use raise to signal an error and provide an optional error message.

- **Handling Exceptions**: Use try, except, else, and finally to handle errors gracefully and perform necessary cleanup.

6.6 User-Defined Exceptions

Creating User-Defined Exceptions

User-defined exceptions are typically created by subclassing the built-in Exception class or one of its subclasses. This allows you to define custom behavior and attributes for your exceptions.
Steps to Create a User-Defined Exception:

1. **Define a New Exception Class**:
    - Subclass from the Exception class (or another built-in exception class if needed).
    - Optionally, override the __init__ method to accept and store additional attributes or messages.

2. **Raise the Custom Exception**:
    - Use the raise keyword to raise your custom exception with any necessary arguments.

3. **Handle the Custom Exception**:
    - Use a try and except block to handle the custom exception.

Example: Creating and Using User-Defined Exceptions

1. Define a Custom Exception Class

```
class MyCustomError(Exception):
    def __init__(self, message, code):
        super().__init__(message)  # Initialize the base Exception class with the message
        self.code = code           # Store additional information like an error code
```

- **__init__ Method**: Initializes the base Exception class with a message and stores additional attributes (e.g., code).

2. Raising the Custom Exception

```
def divide(a, b):
    if b == 0:
        raise MyCustomError("Division by zero is not allowed", 1001)
    return a / b
```

- **raise MyCustomError**: Creates an instance of MyCustomError with a specific message and error code and raises it.

3. Handling the Custom Exception

```
try:
    result = divide(10, 0)
```

```
except MyCustomError as e:
    print(f"Error occurred: {e}")
    print(f"Error code: {e.code}")
```

- **except MyCustomError**: Catches the MyCustomError exception and accesses its message and code attributes.

Detailed Breakdown of User-Defined Exceptions

1. Creating a Custom Exception Class

- **Subclassing Exception**: By inheriting from the Exception class, you gain all the functionality of a standard exception while being able to add your custom features.

```
class MyCustomError(Exception):
    def __init__(self, message, code):
        super().__init__(message)  # Call the base class initializer
        self.code = code           # Add a custom attribute
```

- **Adding Custom Attributes**: You can add custom attributes to store additional information that might be useful when handling the exception.

```
class MyCustomError(Exception):
    def __init__(self, message, code, details=None):
        super().__init__(message)
        self.code = code
        self.details = details  # Optional additional information
```

2. Raising a Custom Exception

- **Usage**: Raise the custom exception in scenarios where specific conditions are met, allowing you to control the flow of your application more precisely.

```
if some_condition:
    raise MyCustomError("An error occurred", 1234)
```

- **Arguments**: You can pass multiple arguments to your custom exception and handle them accordingly.

3. Handling a Custom Exception

- **Accessing Attributes**: When handling the custom exception, you can access the custom attributes you defined.

```
try:
    some_function()
except MyCustomError as e:
    print(f"Message: {e}")
    print(f"Code: {e.code}")
    print(f"Details: {e.details if e.details else 'No details available'}")
```

- **Granularity**: Handling custom exceptions allows for more granular control and specific responses to different error conditions.

Why Use User-Defined Exceptions?

1. **Clarity**: Custom exceptions can make your code more readable and its intent clearer. For example, InvalidUserInputError is more descriptive than a generic Exception.
2. **Control**: They allow for more precise error handling. Instead of catching all errors generically, you can handle specific exceptions uniquely.
3. **Documentation**: Custom exceptions can serve as documentation for the kinds of errors your code might raise, making it easier for others to understand and work with your code.

Example of a Complex Custom Exception
Here's a more comprehensive example with additional attributes and methods:

```
class ValidationError(Exception):
    def __init__(self, message, field, error_code):
        super().__init__(message)
        self.field = field
        self.error_code = error_code

    def __str__(self):
        return f"{self.args[0]} (Field: {self.field}, Error Code: {self.error_code})"

# Usage
def validate_input(data):
    if not data.get('name'):
        raise ValidationError("Name is required", 'name', 400)

try:
    validate_input({})
except ValidationError as e:
    print(f"Validation failed: {e}")
```

In this example:

- **ValidationError Class**: Contains additional attributes and a custom __str__ method for a more informative error message.
- **validate_input Function**: Raises ValidationError with specific information.
- **Handling**: The except block captures the exception and prints detailed information.

Summary

- **User-Defined Exceptions**: Custom exceptions provide a way to handle specific error conditions in your code.
- **Creating**: Subclass the Exception class and define additional attributes and methods if needed.
- **Raising and Handling**: Use raise to trigger and try/except blocks to manage custom exceptions.

Program to handle division by 0

```
try:
```

```
    numerator = float(input("Enter the numerator: "))
    denominator = float(input("Enter the denominator: "))
    result = numerator / denominator
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter numeric values.")
else:
    print(f"The result is {result}")
```

Program to Read from a File Safely

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: The file does not exist.")
else:
    print("File content:")
    print(content)
```

Program to validate user input

```
try:
    number = int(input("Enter an integer: "))
except ValueError:
    print("Error: Invalid input. Please enter a valid integer.")
else:
    print(f"You entered: {number}")
```

Program to Raise and Handle a Custom Exception

```
class NegativeValueError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

def check_positive(value):
    if value < 0:
        raise NegativeValueError("Value cannot be negative.")

try:
    check_positive(-10)
except NegativeValueError as e:
    print(f"Custom error: {e}")
```

Program to Re-raise an Exception

```
def process_value(value):
    try:
        if value < 0:
            raise ValueError("Negative value error.")
    except ValueError as e:
        print(f"Handling in function: {e}")
        raise

try:
    process_value(-5)
except ValueError as e:
    print(f"Caught in main: {e}")
```

UNIT 7
Input and Output in Python & Built in Functions

Topic No.	Topic
7.1	File Objects
7.2	Creating a file object
7.3	Reading file contents
7.4	Writing data into file
7.5	Reading and writing CSV files
7.6	Using with clause

7.7	Using exception handling with file operations

7.1 File Objects

File objects in Python represent an open file on the filesystem. They are instances of the built-in file class and provide methods and attributes to interact with files, such as reading from, writing to, and managing file contents.

Key Concepts:

1. **Creation**: File objects are created using the open() function, which opens a file and returns a file object.

2. **Attributes**:
   - **name**: Returns the name of the file.
   - **mode**: Indicates the mode in which the file was opened (e.g., read, write).
   - **closed**: Indicates whether the file is closed.

3. **Methods**:
   - **Reading Methods**: Methods like read(), readline(), and readlines() are used to retrieve data from the file.
   - **Writing Methods**: Methods like write() and writelines() are used to write data to the file.
   - **File Position**: Methods such as seek() and tell() are used to manage the current file pointer's position.

4. **File Management**:
   - **Closing**: Files should be closed using the close() method to free system resources.
   - **Context Manager**: The with statement is used for automatic file management, ensuring the file is closed after operations.

7.2 Creating a File Object

Definition: Creating a file object involves opening a file and associating it with a file object using the open() function. This file object allows you to perform various operations on the file, such as reading, writing, and closing.

Syntax:
file_object = open(file_name, mode)

Parameters:

- **file_name**: The path to the file you want to open. This can be an absolute path (e.g., '/home/user/file.txt') or a relative path (e.g., 'file.txt').
- **mode**: A string that specifies the mode in which the file should be opened. Common modes include:
  - 'r': Read mode. Opens the file for reading. The file must exist.

- 'w': Write mode. Opens the file for writing. Creates a new file or truncates the existing file.
- 'a': Append mode. Opens the file for appending. Creates a new file if it does not exist.
- 'b': Binary mode. Opens the file in binary mode. Useful for binary files (e.g., 'rb' for reading binary files).
- 'r+': Read and write mode. Opens the file for both reading and writing. The file must exist.

Examples:

1. **Opening a File for Reading**:

file = open("example.txt", "r")
This creates a file object file associated with example.txt in read mode.

2. **Opening a File for Writing**:

file = open("example.txt", "w")
This creates a file object file associated with example.txt in write mode. If the file exists, its contents are truncated.

3. **Opening a File in Binary Mode**:

file = open("example.bin", "rb")
This opens example.bin in binary read mode.

Attributes:

- **name**: Returns the name of the file.

print(file.name)

- **mode**: Returns the mode in which the file was opened.

print(file.mode)

- **closed**: Returns True if the file is closed, False otherwise.

print(file.closed)

Error Handling:

- **Exceptions**: If the file cannot be opened, Python raises exceptions such as FileNotFoundError (file does not exist) and PermissionError (permission denied).

Example:

```
try:
   file = open("example.txt", "r")
except FileNotFoundError:
   print("The file does not exist.")
except PermissionError:
   print("You do not have permission to access the file.")
```

7.3 Reading File Contents

Definition: Reading file contents involves retrieving data from a file using methods provided by the file object. This allows you to access and manipulate the data stored in the file.
Methods:

1. **read(size=-1)**: Reads the entire content of the file or a specified number of bytes. If size is negative, it reads until the end of the file.

content = file.read()

2. **readline(size=-1)**: Reads a single line from the file. If size is specified, it reads up to size bytes.

line = file.readline()

3. **readlines(hint=-1)**: Reads all lines from the file and returns them as a list. The hint parameter specifies an approximate number of bytes to read.

lines = file.readlines()

Examples:

```
# Reading the entire content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)

# Reading line by line
with open("example.txt", "r") as file:
    for line in file:
        print(line)

# Reading all lines as a list
with open("example.txt", "r") as file:
    lines = file.readlines()
    print(lines)
```

7.4 Writing Data into a File

Definition: Writing data into a file involves saving data to a file using methods provided by the file object. This allows you to create new files or modify existing ones by adding or replacing content.

Methods:

1. **write(string)**: Writes a string to the file. If the file is opened in write mode ('w') or append mode ('a'), it writes the string to the file.

file.write("Hello, World!")

2. **writelines(lines)**: Writes a list of strings to the file. Each string is written as-is, without additional newline characters.

file.writelines(["Line 1\n", "Line 2\n"])

Examples:

```
# Writing a string to a file
with open("example.txt", "w") as file:
```

```
    file.write("Hello, World!")

# Writing multiple lines to a file
with open("example.txt", "w") as file:
    lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
    file.writelines(lines)
```

7.5 Reading and Writing CSV Files

Definition: CSV (Comma-Separated Values) files are used to store tabular data. Python's csv module provides tools to read from and write to CSV files.

Reading CSV Files:

1. **Using csv.reader**: Reads rows from a CSV file.

```
import csv
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Writing CSV Files:

1. **Using csv.writer**: Writes rows to a CSV file.

```
import csv
with open("data.csv", "w", newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age", "City"])
    writer.writerow(["John", "30", "New York"])
    writer.writerow(["Alice", "25", "Los Angeles"])
```

Note: Use the newline='' parameter to avoid extra blank lines in CSV files on Windows.

7.6 Using the with Clause

Definition: The with statement simplifies file handling by automatically managing file opening and closing. It ensures that the file is closed after its suite finishes, even if an exception occurs.

Syntax:

```
with open(file_name, mode) as file_object:
    # Perform file operations
```

Example:

```
with open("example.txt", "r") as file:
    content = file.read()
```

```
    print(content)
# File is automatically closed after the block
```

Advantages:
- **Automatic Closure**: Ensures files are properly closed.
- **Error Handling**: Handles exceptions and ensures cleanup.

The with clause in Python is a context manager that simplifies resource management by automatically handling setup and cleanup tasks. When working with files, the with statement ensures that files are properly opened and closed, even if an error occurs during file operations.

1. Purpose of the with Clause
The main purpose of the with clause is to manage resources efficiently and to ensure that resources are released properly after their use. It helps avoid common issues such as forgetting to close a file, which can lead to resource leaks and other problems.

Benefits:
- **Automatic Resource Management**: Ensures resources are properly cleaned up.
- **Simplifies Code**: Reduces the amount of boilerplate code for opening and closing resources.
- **Error Handling**: Manages resources even if an exception occurs.

2. Syntax of the with Clause
The syntax for using the with clause is as follows:

```
with expression as variable:

    # Code block where the resource is used
```

- **expression**: The expression evaluates to a context manager. For file operations, this is typically the open() function.
- **variable**: The file object or other resource that you will use within the with block.
- **Code block**: The block of code that performs operations using the resource.

3. Example: Using with for File Operations
Opening a File for Reading:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# File is automatically closed here
```

In this example:
- open("example.txt", "r") returns a file object.
- file is the variable that refers to this file object.
- The file is automatically closed after the with block, even if an exception occurs.
-

Opening a File for Writing:

```
with open("example.txt", "w") as file:
```

```
    file.write("Hello, World!")
# File is automatically closed here
```

In this example:

- The file example.txt is opened in write mode ("w").
- The string "Hello, World!" is written to the file.
- The file is automatically closed after the with block.

4. Working with Other Context Managers

The with statement is not limited to file handling; it can be used with other resources like database connections, locks, and more. Any object that implements the context management protocol (i.e., has __enter__ and __exit__ methods) can be used with the with statement.

Example: Using with for Database Connections:

```
import sqlite3

with sqlite3.connect("example.db") as connection:
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM table_name")
    results = cursor.fetchall()
    print(results)
# Database connection is automatically closed here
```

5. How with Works

1. **Entering the Context**: When the with statement is executed, the __enter__ method of the context manager is called. This method sets up the resource and returns it, assigning it to the variable specified in the with statement.
2. **Executing the Block**: The code block within the with statement is executed using the resource.
3. **Exiting the Context**: After the code block is executed, the __exit__ method of the context manager is called. This method performs cleanup, such as closing the file or releasing other resources.

Example with Custom Context Manager:

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")

with MyContextManager() as cm:
    print("Inside the context")
# Output:
# Entering the context
# Inside the context
# Exiting the context
```

In this custom context manager:

- The __enter__ method is called when entering the context.
- The __exit__ method is called when exiting the context.

6. Common Errors and Pitfalls

- **Missing __exit__ Method**: If a custom context manager does not implement the __exit__ method, it will not properly clean up resources.
- **Exceptions in __enter__**: If an exception occurs in the __enter__ method, the context manager will not be entered, and the __exit__ method will not be called.

Summary

- The with clause is used to manage resources automatically and ensure proper cleanup.
- It simplifies code and reduces the risk of resource leaks.
- It works with any object that implements the context management protocol (__enter__ and __exit__ methods).
- It can be used for various resources beyond file handling, such as database connections and locks.

Using the with clause is a best practice for managing resources in Python, helping to write cleaner, more reliable code.

7.7 Using Exception Handling with File Operations

Definition: Exception handling with file operations involves managing errors that may occur during file handling. Common exceptions include file not found, permission issues, and I/O errors.

Using try, except, and finally:

1. **Basic Handling**:

```
try:
   file = open("example.txt", "r")
   content = file.read()
except FileNotFoundError:
   print("File not found.")
except PermissionError:
   print("Permission denied.")
finally:
   file.close()  # Ensures file is closed
```

2. **Using with Statement**:

```
try:
   with open("example.txt", "r") as file:
      content = file.read()
except FileNotFoundError:
   print("File not found.")
except PermissionError:
   print("Permission denied.")
```

Common Exceptions:

- **FileNotFoundError**: Raised when the file is not found.
- **PermissionError**: Raised when there is a permission issue.
- **IOError**: Raised for I/O related errors.

MCQs

1. **Which of the following is used to output data to the console in Python?**
   - A) print()
   - B) write()
   - C) echo()
   - D) output()

   Answer: A) print() **Explanation:** The print() function is used to display output to the console.

2. **What is the correct file extension for Python files?**
   - A) .pyth
   - B) .python
   - C) .py
   - D) .pyt

   Answer: C) .py **Explanation:** Python files use the .py extension.

3. **Which of the following is a valid variable name in Python?**
   - A) 1variable
   - B) variable_name
   - C) variable name
   - D) @variable

   Answer: B) variable_name **Explanation:** Variable names in Python must start with a letter or underscore and can include letters, digits, and underscores.

4. **Which of the following is the correct syntax for an if statement in Python?**
   - A) if condition:
   - B) if condition then:
   - C) if condition;
   - D) if (condition) {

   Answer: A) if condition: **Explanation:** The correct syntax for an if statement includes a colon at the end of the condition.

5. **What will be the output of the following code? print(3 * 'ha')**
   - A) hahaha
   - B) ha ha ha
   - C) 3ha
   - D) 3*ha

   Answer: A) hahaha **Explanation:** Multiplying a string by an integer repeats the string that many times.

6. **Which keyword is used to exit a loop in Python?**
   - A) exit
   - B) break
   - C) stop
   - D) end

   Answer: B) break **Explanation:** The break keyword exits the loop immediately.

7. **How do you access the third element in a list my_list?**
   - A) my_list[2]
   - B) my_list[3]
   - C) my_list.get(2)
   - D) my_list(2)

   Answer: A) my_list[2] **Explanation:** List indexing in Python starts at 0, so the third element is accessed with index 2.

8.**Which method is used to add an element at the end of a list?**
   - A) append()
   - B) insert()
   - C) extend()
   - D) add()

   Answer: A) append() **Explanation:** The append() method adds an element to the end of a list.

9. **What does the range(5, 10) function call return?**
   - A) Numbers from 5 to 10 inclusive
   - B) Numbers from 5 to 9 inclusive
   - C) Numbers from 10 to 5
   - D) Numbers from 5 to 10 exclusive

   Answer: B) Numbers from 5 to 9 inclusive **Explanation:** range(5, 10) generates numbers starting from 5 up to but not including 10.

10.**How do you create a tuple in Python?**
   - A) my_tuple = [1, 2, 3]
   - B) my_tuple = (1, 2, 3)
   - C) my_tuple = {1, 2, 3}
   - D) my_tuple = 1, 2, 3

   Answer: B) my_tuple = (1, 2, 3) **Explanation:** Tuples are defined with parentheses.

11. **Which of the following is a mutable data structure?**
   - A) tuple
   - B) list
   - C) set
   - D) string

   Answer: B) list **Explanation:** Lists are mutable, meaning they can be changed after creation.

12 **What will be the output of len({'a': 1, 'b': 2})?**
   - A) 2
   - B) 1
   - C) 3
   - D) 0

   Answer: A) 2 **Explanation:** len() returns the number of items in the dictionary, which is 2.

13. **How do you retrieve a value from a dictionary using a key?**

- A) dict[key]
- B) dict.get(key)
- C) Both A and B
- D) dict.retrieve(key)

  Answer: C) Both A and B **Explanation:** Values can be retrieved using both dict[key] and dict.get(key).

14. **Which method is used to remove the last item from a list?**
    - A) pop()
    - B) remove()
    - C) del()
    - D) discard()

      Answer: A) pop() **Explanation:** The pop() method removes and returns the last item from the list.

15. **What will be the result of set([1, 1, 2, 2, 3, 3])?**
    - A) {1, 2, 3}
    - B) {1, 2, 3, 1, 2, 3}
    - C) [1, 2, 3]
    - D) Error

      Answer: A) {1, 2, 3} **Explanation:** Sets automatically remove duplicate values.

16. **Which method is used to combine two lists?**
    - A) append()
    - B) extend()
    - C) join()
    - D) combine()

      Answer: B) extend() **Explanation:** The extend() method appends the elements of one list to another.

17. **How do you handle an exception in Python?**
    - A) try and catch
    - B) try and except
    - C) try and finally
    - D) try and raise

      Answer: B) try and except **Explanation:** The try and except block is used to handle exceptions.

18. **What is the purpose of the finally block in exception handling?**
    - A) To catch exceptions
    - B) To execute code regardless of exceptions
    - C) To define an exception
    - D) To raise an exception

      Answer: B) To execute code regardless of exceptions **Explanation:** The finally block is executed no matter if an exception occurs or not.

19. **Which of the following is used to import a module in Python?**
    - A) include
    - B) import

- C) require
- D) use

  Answer: B) import **Explanation:** The import statement is used to include modules in Python.

20. **Which method is used to read a file's content in Python?**
    - A) file.read()
    - B) file.open()
    - C) file.write()
    - D) file.close()

      Answer: A) file.read() **Explanation:** The read() method is used to read the contents of a file.

21. **How do you define a function in Python?**
    - A) function name():
    - B) def name():
    - C) func name():
    - D) define name():

      Answer: B) def name(): **Explanation:** Functions are defined using the def keyword followed by the function name and parentheses.

22. **Which method is used to handle errors in Python code?**
    - A) try
    - B) except
    - C) catch
    - D) finally

      Answer: B) except **Explanation:** The except block is used to catch and handle exceptions.

23. **What is the output of print("Hello".upper())?**
    - A) HELLO
    - B) hello
    - C) Hello
    - D) Error

      Answer: A) HELLO **Explanation:** The upper() method converts all characters to uppercase.

24. **Which keyword is used to create a class in Python?**
    - A) class
    - B) def
    - C) create
    - D) object

      Answer: A) class **Explanation:** The class keyword is used to define a class.

25. **How do you call a method from a class?**
    - A) class.method()
    - B) object.method()
    - C) method()
    - D) call.method()

Answer: B) object.method() **Explanation:** Methods are called on an instance of the class, i.e., object.method().

26. **What is the use of the lambda function in Python?**
    - A) To create anonymous functions
    - B) To define a class
    - C) To create a loop
    - D) To handle exceptions

    Answer: A) To create anonymous functions **Explanation:** lambda functions are used to create small, anonymous functions.

27. **Which function is used to get the number of items in a dictionary?**
    - A) count()
    - B) len()
    - C) size()
    - D) length()

    Answer: B) len() **Explanation:** The len() function returns the number of items in a dictionary.

28. **What is the purpose of the self keyword in Python classes?**
    - A) To refer to the class itself
    - B) To refer to the instance of the class
    - C) To create a new class
    - D) To handle errors

    Answer: B) To refer to the instance of the class **Explanation:** self refers to the instance of the class within methods.

29. **What is the default return value of a function that does not return a value explicitly?**
    - A) 0
    - B) None
    - C) False
    - D) Error

    Answer: B) None **Explanation:** If no return value is specified, the function returns None by default.

30. **Which of the following is used to handle multiple exceptions in a single block?**
    - A) except (Exception1, Exception2)
    - B) except Exception1 or Exception2
    - C) except [Exception1, Exception2]
    - D) except (Exception1 | Exception2)

    Answer: A) except (Exception1, Exception2) **Explanation:** Multiple exceptions can be handled by specifying them as a tuple in the except block.

Internal Examination (May/June-2023)

Course: BCA **Semester: V**
Subject: Python **CourseCode:**
Max. Marks: 40

Max. Time: 1:30 hrs

Instructions: Use of calculator for subjects like Financial Management, operation etc. allowed if required. (Scientific calculators not allowed).
Use of unfair means will lead to cancellation of paper followed by disciplinary action.

Attempt any two questions from section-I and Attempt any two question from section-II.

Section-I

(Theoretical Concept and Practical/Application oriented)

Answer in 500 words. Each question carries 08 marks.

Q1.

Q2.

Q3.

Q4. Write short note on any two. Answer in 300 words. **Each carries 04 marks.**

a)
b)
c)

Section-II

(Analytical Question / Case Study / Essay Type Question to test analytical and Comprehensive Skills)

Answer in 700 words. Attempt any 2 questions. Each question carries 12 marks

Q5.

Q6.

Q7.

List of Python Programming Exercises

1. Python List Exercises
2. Python program to interchange first and last elements in a list
3. Python program to swap two elements in a list
4. Python | Ways to find length of list
5. Maximum of two numbers in Python
6. Minimum of two numbers in Python

Python String Exercises

1. Python program to check whether the string is Symmetrical or Palindrome
2. Reverse words in a given String in Python
3. Ways to remove i'th character from string in Python
4. Find length of a string in python (4 ways)
5. Python program to print even length words in a string

Python Tuple Exercises

1. Python program to Find the size of a Tuple
2. Python – Maximum and Minimum K elements in Tuple
3. Python – Sum of tuple elements
4. Python – Row-wise element Addition in Tuple Matrix
5. Create a list of tuples from given list having number and its cube in each tuple

Python Dictionary Exercises

1. Python | Sort Python Dictionaries by Key or Value
2. Handling missing keys in Python dictionaries
3. Python dictionary with keys having multiple inputs
4. Python program to find the sum of all items in a dictionary
5. Python program to find the size of a Dictionary

Python Set Exercises

1. Find the size of a Set in Python
2. Iterate over a set in Python
3. Python – Maximum and Minimum in a Set
4. Python – Remove items from Set

Python – Check if two lists have atleast one element common

Python Matrix Exercises

1. Python – Assigning Subsequent Rows to Matrix first row elements
2. Adding and Subtracting Matrices in Python

3. Python – Group similar elements into Matrix
4. Python – Row-wise element Addition in Tuple Matrix
5. Create an n x n square matrix, where all the sub-matrix have the sum of opposite corner elements as even

Python Functions Exercises

1. How to get list of parameters name from a function in Python?
2. How to Print Multiple Arguments in Python?
3. Python program to find the power of a number using recursion
4. Sorting objects of user defined class in Python
5. Functions that accept variable length key value pair as arguments

1. Python Lambda Exercises
2. Lambda with if but without else in Python
3. Python | Sorting string using order defined by another string
4. Python | Find fibonacci series upto n using lambda
5. Python program to count Even and Odd numbers in a List
6. Python | Find the Number Occurring Odd Number of Times using Lambda expression and reduce function

1. Python Pattern printing Exercises
2. Program to print half Diamond star pattern
3. Programs for printing pyramid patterns in Python
4. Program to print the diamond shape
5. Python | Print an Inverted Star Pattern
6. Python Program to print digit pattern

Python DateTime Exercises

1. Python program to get Current Time
2. Get Yesterday's date using Python
3. Python program to print current year, month and day
4. Python – Convert day number to date in particular year
5. Get Current Time in different Timezone using Python

Python OOPS Exercises

1. Python program to build flashcard using class in Python
2. Shuffle a deck of card with OOPS in Python
3. How to create an empty class in Python?
4. Student management system in Python

Python Regex Exercises

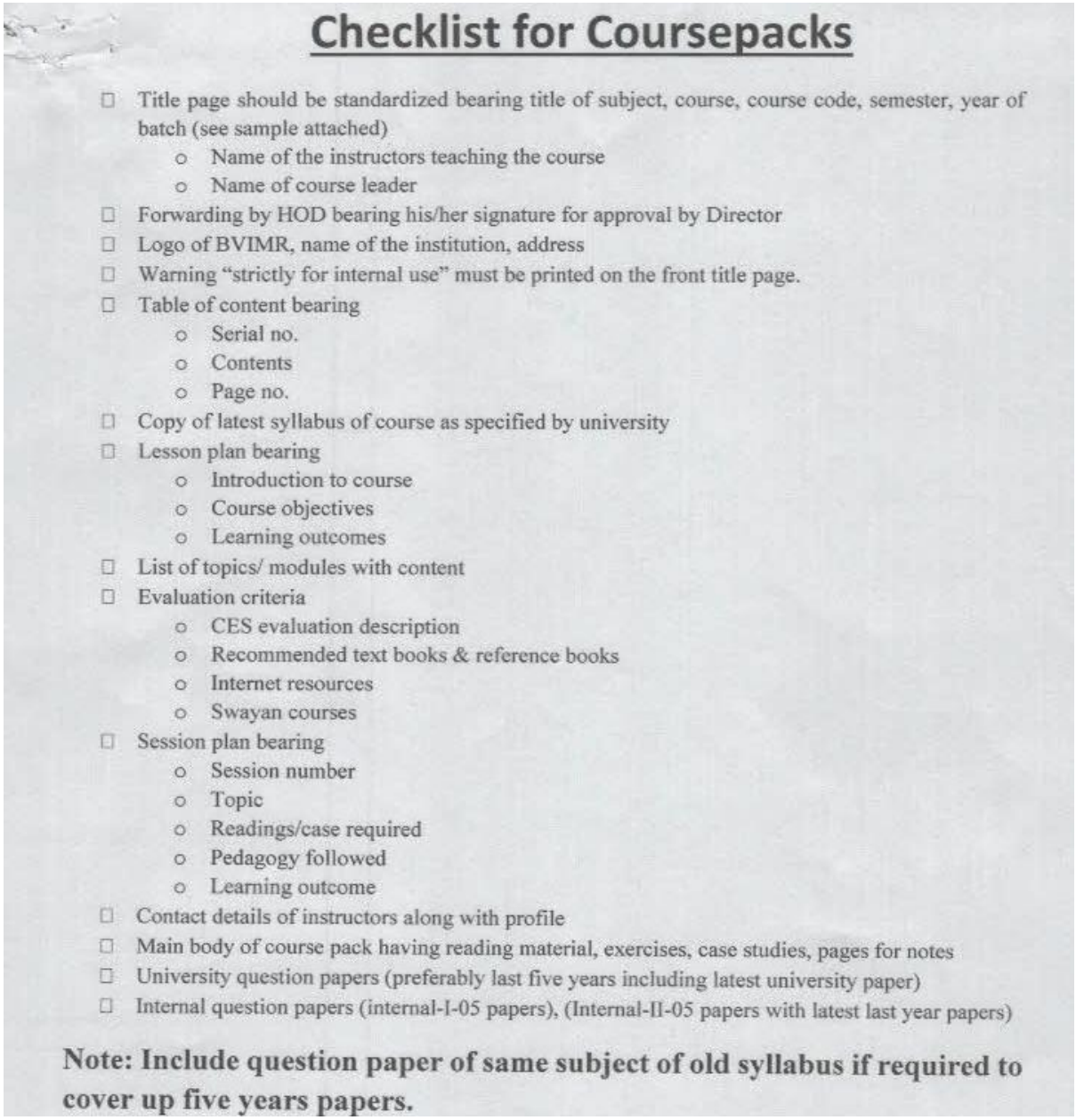
1. Python program to find the type of IP Address using Regex
2. Python program to find Indices of Overlapping Substrings
3. Python program to extract Strings between HTML Tags
4. Python – Check if String Contain Only Defined Characters using Regex
5. Python program to find files having a particular extension using RegEx

Python File Handling Exercises

1. Read content from one file and write it into another file
2. Write a dictionary to a file in Python
3. How to check file size in Python?
4. Find the most repeated word in a text file
5. How to read specific lines from a File in Python?

Python CSV Exercises

1. Update column value of CSV in Python
2. How to add a header to a CSV file in Python?
3. Get column names from CSV using Python
4. Writing data from a Python List to CSV row-wise
5. Convert multiple JSON files to CSV Python



Checklist for Coursepacks

- ☐ Title page should be standardized bearing title of subject, course, course code, semester, year of batch (see sample attached)
  - Name of the instructors teaching the course
  - Name of course leader
- ☐ Forwarding by HOD bearing his/her signature for approval by Director
- ☐ Logo of BVIMR, name of the institution, address
- ☐ Warning "strictly for internal use" must be printed on the front title page.
- ☐ Table of content bearing
  - Serial no.
  - Contents
  - Page no.
- ☐ Copy of latest syllabus of course as specified by university
- ☐ Lesson plan bearing
  - Introduction to course
  - Course objectives
  - Learning outcomes
- ☐ List of topics/ modules with content
- ☐ Evaluation criteria
  - CES evaluation description
  - Recommended text books & reference books
  - Internet resources
  - Swayan courses
- ☐ Session plan bearing
  - Session number
  - Topic
  - Readings/case required
  - Pedagogy followed
  - Learning outcome
- ☐ Contact details of instructors along with profile
- ☐ Main body of course pack having reading material, exercises, case studies, pages for notes
- ☐ University question papers (preferably last five years including latest university paper)
- ☐ Internal question papers (internal-I-05 papers), (Internal-II-05 papers with latest last year papers)

Note: Include question paper of same subject of old syllabus if required to cover up five years papers.

Declaration by Faculty:

I, Nripesh Kumar Nrip, Dr. Mansi Agnihotri Designation asst. professor, Visiting faculty Teaching Python Programming subject in BCA course V Sem have incorporated all the necessary pages section/quotations papers mentioned in this check list above.

Signature