Python Quick Reference

Essential Syntax and Concepts

import this

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
numbers = [1, 2, 3, 4, 5]
squared_numbers = [x**2 for x in numbers]
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6, 8, 10]

# Python 3: Fibonacci series up to n
def fib(n):
    a,b = 0,1
    while a < n:
        print(a, end=' ')
        a,b = b,a+b
    print()
fib(10)
#Output: 0 1 1 2 3 5 8
```

PALGUNI G T

Literature International Publishers
IIP

Python Quick Reference

Essential Syntax and Concepts

First Edition

Author
Palguni G. T

Title of the Book: Python Quick Reference Essential Syntax and Concepts

Edition First: 2023

Disclaimer

The author is solely responsible for the contents published in this book. The publishers or editors do not take any responsibility for the same in any manner. Errors, if any, are purely unintentional and readers are requested to communicate such errors to the editors or publishers to avoid discrepancies in future.

ISBN: 978-93-5747-903-5

MRP Rs. 440/-

Publisher, Printed at & Distribution by:
Selfypage Developers Pvt Ltd.,
Pushpagiri Complex,
Beside SBI Housing Board,
K.M. Road Chikkamagaluru, Karnataka.
Tel.: +91-8861518868
E-mail:publish@iiponline.org

IMPRINT: I I P Iterative International Publishers

For Sales Enquiries
Contact: +91- 8861511583
E-mail: sales@iiponline.org

Dedicated To

My Parents and Friends

Preface....

Welcome to "**Python Quick Reference: Essential Syntax and Concepts**". This comprehensive and concise guide is designed to be your go-to resource for mastering Python, one of the most versatile and widely used programming languages in the world.

Python's simplicity and readability make it an excellent choice for both beginners and experienced developers. Whether you're just starting your programming journey or looking to expand your skills, this book provides a valuable reference for understanding Python's core syntax and concepts.

Book Structure

- **Introduction to Python Programming Language:** I begin by introducing Python, delving into its history and its significance within the programming world.
- **Zen of Python:** Delve into the guiding principles and philosophies that shape Python development.
- **Essential Syntax and Concepts:** Gain a solid foundation by exploring Python's program structure, symbols, tokens, keywords, and more.
- **Data Handling:** Dive into the world of data in Python, including variables, data types, operators, and control statements.
- **Functions and Modules:** Learn about functions, scope, and modules, essential for building structured and reusable code.
- **Working with Data Structures:** Explore Python's built-in data structures, including lists, dictionaries, sets, and tuples.
- **Advanced Topics:** Go beyond the basics with topics like exception handling, file I/O, object-oriented programming, regular expressions, web scraping, and more.
- **Data Science and Visualization:** Get an introduction to data science libraries like NumPy, Pandas, Matplotlib, and Seaborn, essential for data analysis and visualization.
- **GUI Development**: Discover the world of graphical user interface (GUI) development using the Tkinter library.
- **Interview Questions and Sample Programs:** Prepare for interviews with a collection of Python interview questions and explore sample Python programs.
- **Tips and Tricks:** Learn valuable tips and tricks to enhance your Python programming skills.
- **References:** Find additional resources for further exploration and learning.

"**Python Quick Reference**" is designed to be a practical and accessible guide that you can turn to whenever you need to refresh your memory on Python syntax or concepts. Whether you're a student, a professional developer, or anyone looking to harness the power of Python, this book is your essential companion on your Python journey.

I authored this book during my summer vacation in the months of October and November 2023 as a passionate endeavor. To prepare for this project, I diligently completed prerequisite courses, including "Introduction to Python Programming" during my regular B.E 1st Semester and "Python for Data Science" through an NPTEL Course.

The concepts and programs featured in this book have been meticulously crafted through an in-depth study and analysis of the literature mentioned in the references section. Additionally, ChatGPT-3.5 prompts were employed during the design process. These programs were executed and rigorously tested within the Jupyter Notebook environment to ensure their effectiveness and reliability.
I hope you will find this book to be a valuable resource as you navigate the world of Python programming. Happy coding!

Palguni G T
Author

Acknowledgements....

I express my gratitude to my father, Dr. Thyagaraju G S, who is currently working as a professor and HoD in the Department of CSE at SDM Institute of Technology, Ujire. His invaluable assistance in designing the book is deeply appreciated. His contributions included setting the Contents and reviewing the book.

……..**Palguni G T**

Contents....

1

Introduction to Python Programming Language

Python is a high-level programming language that has gained widespread popularity due to its simplicity, readability, and versatility. Guido van Rossum released the first version of Python in 1991, and since then, it has evolved into a powerful tool used in various domains, from web development to data science.
List of **key** features of the Python programming language:

Readability and Simplicity: Python's clean and intuitive syntax emphasizes code readability, reducing the chances of errors and making it easier for developers to write and maintain code.

Interpreted: Python code is executed line by line by an interpreter, eliminating the need for a separate compilation step.

Dynamic Typing: Variables are dynamically typed, meaning their types are determined during runtime.

Large Standard Library: Python comes with an extensive collection of built-in modules and libraries that provide ready-to-use functions and tools for various tasks, saving development time**.**

Extensibility: Python can be easily integrated with other languages like C, C++, and Java, allowing for optimized performance or interaction with existing codebases.

Indentation-Based Blocks: Python uses indentation to define code blocks, promoting consistent and well-structured code.

Object-Oriented: Python supports object-oriented programming, allowing developers to model real-world entities with classes and objects.

Open Source: Python is an open-source language, which means the source code is available to the public and can be modified and distributed freely.

Portability: Python code written on one platform can run on other platforms with minimal modifications.

Versatility: Python can be used in various domains, including web development, data analysis, scientific computing, machine learning, artificial intelligence, automation, scripting, and more, making it a versatile language for diverse applications.

Cross-Platform Compatibility: Python is available on multiple platforms and operating systems, ensuring that code can be written once and run on different environments.

High-Level Language: Python abstracts complex low-level operations, providing a higher-level interface and allowing developers to focus on problem-solving.

Diverse Community and Ecosystem: Python has a large and active community of developers, resulting in extensive resources, support, and a wide range of third-party libraries and frameworks.

Exception Handling: Python provides a robust and clear mechanism for handling exceptions, allowing developers to write code that gracefully handles errors and exceptions.

2

Zen of Python

The "**Zen of Python**" is a collection of guiding principles for writing computer programs in the Python language. These principles are meant to encapsulate the philosophy and design principles that shape the Python programming community's approach to software development. They were written by **Tim Peters**, a prominent Python developer, in 2004 and are accessible by typing **import this** in a Python interpreter or script.

Code :

```
import this
```

#Output

The Zen of Python, by Tim Peters

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one-- and preferably only one --obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea -- let's do more of those!

3

Structure of Python Program

A typical Python program consists of different components and follows a specific structure. Here's an explanation of each part along with an example:

```
# Import statements (optional)
import math
import random

# Function and class definitions (optional)
def greet(name):
    return f"Hello, {name}!"

class Calculator:
    def add(self, a, b):
        return a + b

# Main code block
def main():
    user_name = input("Enter your name: ")
    greeting = greet(user_name)
    print(greeting)

    calculator = Calculator()
    result = calculator.add(5, 7)
    print("Result:", result)

# Conditional check to ensure the script is run as the main program
if __name__ == "__main__":
    main()
```

Import Statements: These lines bring in external modules or libraries to use their functions and classes. They're usually placed at the beginning of the script.

Function and Class Definitions: This is where you define functions and classes that you'll use in the main code block. These definitions provide a way to encapsulate and organize your code.

Main Code Block: This is the core of your program where you put the main logic and operations. It's the part that gets executed when you run the script.

Conditional Check (if name == "main"): This conditional statement checks whether the script is being run as the main program (not imported as a module). This ensures that the main code block is executed only when the script is run directly.

4

Python Symbols

Python uses a variety of symbols and operators for different purposes. There are over 40 symbols and operators in Python. This includes mathematical operators, comparison operators, logical operators, assignment operators, bitwise operators, punctuation symbols, and more. The exact number can vary slightly depending on how you categorize certain operators.

Sl. No	Symbols	Meaning
1.	+	Addition
2.	-	Subtraction
3.	*	Multiplication
4.	/	Division
5.	%	Modulo (remainder)
6.	**	Exponentiation
7.	//	Floor division (quotient without remainder)
8.	=	Assignment
9.	==	Equal to
10.	!=	Not equal to
11.	<	Less than
12.	>	Greater than
13.	<=	Less than or equal to
14.	>=	Greater than or equal to
15.	**and**	Logical AND
16.	**or**	Logical OR
17.	**not**	Logical NOT
18.	**is**	Identity comparison
19.	**in**	Membership test
20.	**not in**	Negated membership test
21.	+=	Add and assign
22.	-=	Subtract and assign
23.	*=	Multiply and assign
24.	/=	Divide and assign
25.	%=	Modulo and assign
26.	**=	Exponentiation and assign
27.	//=	Floor division and assign
28.	&	Bitwise AND

29.	**|**	Bitwise OR
30.	^	Bitwise XOR
31.	**~**	Bitwise NOT
32.	<<	Bitwise left shift
33.	>>	Bitwise right shift
34.	**()**	Parentheses
35.	**[]**	Brackets
36.	**{}**	Curly braces
37.	**:**	Colon
38.	**,**	Comma
39.	**.**	Dot
40.	**...**	Ellipsis
41.	->	Function annotation (used to indicate the return type)
42.	**@ :**	Decorator symbol

5

Python Tokens

In programming languages, a token is a basic building block, representing the smallest individual units of code. Tokens are used by the compiler or interpreter to understand and process the code. In Python, tokens include identifiers, keywords, literals, operators, and other symbols. Here is a list of Python tokens:

1. **Keywords:** And, as, assert, async, await, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield.

2. **Identifiers:** Names given to various programming elements such as variables, functions, classes, etc.

   Examples: variable_name, functionName, ClassName

3. **Literals**

   Numeric Literals:
   Integers: 42, -100, 0b1010 (binary), 0o755 (octal), 0x1F (hexadecimal)
   Floating-Point: 3.14, -0.5, 1e3 (scientific notation)

   String Literals
   'Hello, world!'
   "Python"
   '''Multiline string'''

   Boolean Literals: True, False

   Non-Literal: None

4. **Operators**

   Arithmetic Operators: +, -, *, /, %, //, **
   Comparison Operators: ==, !=, <, >, <=, >=
   Assignment Operators: =, +=, -=, *=, /=, %=, //=, **=
   Bitwise Operators: &, |, ^, ~, <<, >>

Logical Operators: and, or, not, is, in, not in
Membership Operators: in, not in
Identity Operators: is, is not

5. **Delimiters**
   (,) : Parentheses
   [,] : Brackets
   {, } : Curly Braces
   , : Comma
   : : Colon
   ; : Semicolon

6. **Comments:** Lines beginning with # that provide explanatory text.

7. **Special tokens:** ... (ellipsis), None (null-like object), True, False.

8. **String formatting tokens:** % (used with strings to format values).

These are the main categories of Python tokens. They make up the syntax of the Python programming language and are used to compose valid code that the interpreter or compiler can understand and execute.

6

Python Keywords

Keywords in Python are reserved words that have special meanings and purposes within the programming language. These keywords cannot be used as identifiers (variable names, function names, etc.) because they are already predefined for specific operations or control flow. Examples of Python keywords include if, else, while, for, def, class, import, return, and so on.

To list all the keywords in Python, you can use the keyword module. Here's a code snippet that demonstrates how to do this:

```
import keyword
# Get a list of all keywords
keywords = keyword.kwlist
# Print the list of keywords
for kw in keywords:
    print(kw)
```

List of all 35 Python Keywords

Sl. No	Keyword	Meaning
1.	False	Boolean value representing false.
2.	None	Represents a null or empty value.
3.	True	Boolean value representing true.
4.	and	Logical operator for boolean AND.
5.	as	Used in context of the with statement to create an alias.
6.	assert	Used for debugging, raises an exception if a given condition is not true.
7.	async	Declares a coroutine function.
8.	await	Used to pause the execution of a coroutine until a result is ready.
9.	break	Used to exit the current loop.
10.	class	Defines a class.
11.	continue	Used to skip the rest of the current loop iteration and move to the next one.
12.	def	Defines a function.

13.	del	Used to delete references to objects.
14.	elif	Short for "else if", used in conditional statements.
15.	else	Executes when the conditional statement is false.
16.	except	Catches exceptions in a try...except block.
17.	finally	Executes code regardless of whether an exception occurred or not.
18.	for	Used to iterate over a sequence (like a list or string).
19.	from	Used to import specific attributes or functions from a module.
20.	global	Declares that a variable is global, not local.
21.	if	Used for conditional branching.
22.	import	Used to import modules into a Python script.
23.	in	Used to check if a value exists in a sequence.
24.	is	Used to compare object identity.
25.	lambda	Used to create anonymous (inline) functions.
26.	nonlocal	Declares that a variable refers to a variable in an outer (but non-global) scope.
27.	not	Logical operator for boolean NOT.
28.	or	Logical operator for boolean OR.
29.	pass	Placeholder statement that does nothing, used for code structure.
30.	raise	Raises an exception explicitly.
31.	return	Used to return a value from a function.
32.	try	Begins a block of code that might raise exceptions.
33.	while	Creates a loop that executes as long as a given condition is true.
34.	with	Used to simplify exception handling by encapsulating resource management.
35.	yield	Used in generator functions to produce a value for iteration.

7

Expressions

In programming, an expression is a combination of one or more values, variables, operators, and function calls that are evaluated to produce a result. Expressions are the building blocks of statements and are used to perform calculations, comparisons, and other operations in a programming language. An expression can be as simple as a single value or as complex as a combination of values and operations.

Here are a few examples of expressions

Type of Expression	Example Expression	Result
Arithmetic	5 + 3	8
	2 * 4 - 1	7
	10 / 2	5.0
	9 // 2	4
	3 ** 2	9
	7 % 3	1
String Concatenation	"Hello" + " " + "World"	"Hello World"
	'Python' * 3	"PythonPythonPython"
Comparison	7 > 5	True
	10 == 5	False
	x != y	True
	age >= 18 and age <= 65	Depends on age value
Logical	True and False	False
	not True	False
Function Call	len("Hello")	5
	math.sqrt(25)	5.0 (if math module is imported)
	my_function(2, 3)	Depends on my_function
Conditional (Ternary)	x if x > 0 else -x	Depends on x value
List Comprehension	[x * 2 for x in range(5)]	[0, 2, 4, 6, 8]
Set Comprehension	{x % 2 for x in range(10)}	{0, 1}
Dictionary Comprehension	{x: x ** 2 for x in range(3)}	{0: 0, 1: 1, 2: 4}

Generator Expression	(x ** 2 for x in range(5))	Generator object
Lambda	lambda x: x ** 2	Lambda function object
Attribute Access	my_object.attribute	Value of attribute
	module.constant	Value of constant
	my_list.append(5)	Appends 5 to my_list
Indexing	my_list[2]	Value at index 2
	my_string[1:4]	Substring from index 1 to 3
	my_tuple[-1]	Last value of the tuple
	my_dict['key']	Value associated with 'key'
	matrix[2][3]	Value in row 2, column 3
Bitwise	5 & 3	1 (bitwise AND)
	~12	-13 (bitwise NOT)
	10 ^ 3	9 (bitwise XOR)
	16 >> 2	4 (bitwise right shift)
	4 << 1	8 (bitwise left shift)

Please note that some expressions depend on the values of variables or functions used, and the examples provided are illustrative.

8

Comments

In programming, comments are annotations or explanatory notes added to the source code that are ignored by the compiler or interpreter. Comments are used to provide information, explanations, or context about the code to developers, making it easier to understand and maintain. They are not executed as part of the program and do not affect the program's functionality.

Comments are crucial for documenting code, improving its readability, and helping other developers (including your future self) understand the purpose and logic behind the code.

In Python, there are two types of comments:

Single-line Comments:
Single-line comments are used to add a short description to a single line of code. In Python, single-line comments start with the # symbol and continue until the end of the line.

```
# This is a single-line comment
x = 5  # This is also a comment explaining the variable assignment
```

Multi-line Comments (Docstrings):
While Python doesn't have a traditional multi-line comment syntax like some other languages, it uses docstrings (documentation strings) to create multi-line comments. Docstrings are often used to document functions, classes, and modules. They are enclosed in triple quotes (''' or """) and can span multiple lines.

```
def add(x, y):
    """
    This function takes two arguments, x and y,
    and returns their sum.
    """
    return x + y
```

The docstring in this example provides a detailed explanation of the purpose and behavior of the add function.

Comments play a crucial role in making code more understandable, especially for complex algorithms or when collaborating with other programmers. They help in providing context, explaining decisions, and highlighting important details about the code. However, it's important to maintain a balance – while comments are useful, well-written code should also be self-explanatory and easy to understand without excessive comments.

Benefits of Comments: Comments in programming offer several benefits, enhancing the quality, maintainability, and collaborative aspects of code development. Some of the key benefits of using comments include:

Code Documentation: Comments serve as a form of documentation, explaining the purpose, behavior, and usage of code elements such as functions, classes, and variables. They provide a clear understanding of what each part of the code is meant to do.

Explanation of Logic: Comments can help explain complex algorithms, intricate logic, or non-obvious solutions. This is especially valuable when dealing with intricate business rules or optimization techniques that might not be immediately apparent from the code alone.

Maintainability: Well-commented code is easier to maintain. If another developer, including your future self, needs to modify or extend the code later on, comments provide insights into how the code works and what its intended behavior is.

Code Reviews: During code reviews, comments can aid in discussing design decisions, suggesting improvements, or pointing out potential issues. They facilitate communication between team members and ensure that everyone is on the same page.

Collaboration: In collaborative projects, where multiple developers work together, comments help other team members understand your code and contribute effectively. Clear comments reduce the learning curve for new team members joining the project.

Debugging and Troubleshooting: Comments can be used to indicate areas of code that might be problematic or need further testing. They can also provide insights into known issues or workarounds.

Regulatory and Compliance: In industries with regulatory requirements, comments can document compliance strategies, security measures, or data handling procedures.

Educational Purposes: Comments can act as teaching tools, helping novice programmers learn about different programming concepts by reading and understanding code.

Code Reuse: Comments can include information about how to use or modify a piece of code for reuse in other projects.

Version Control and History: Comments in commit messages or version control annotations help track changes, making it easier to understand the purpose of each modification over time.

While comments offer numerous advantages, it's important to use them judiciously and effectively. Over-commenting can clutter code and make it harder to read. Ideally, code should be written in a self-explanatory manner, and comments should complement the code by providing additional context, explanations, or insights.

9

Indentation

Indentation in programming refers to the practice of structuring code by visually aligning blocks of code to indicate their hierarchy or scope. It is primarily used in languages that use indentation for block delimiters, like Python. Indentation is crucial for code readability and to define the scope of control structures like loops, conditionals, and functions.

In Python, indentation is significant because it replaces traditional braces or keywords to mark code blocks. It enforces a consistent and clear way of representing the nesting and structure of code. Indentation typically consists of **spaces or tabs**, but they must be used consistently within a single code block.

Here's an example to illustrate indentation in Python:

```
if True:
    print("This line is indented by one level.")
    if False:
        print("This line is indented by two levels.")
    print("This line is also indented by one level.")
print("This line is at the outermost level.")
```

In this example

- **The if True:** statement is a block of code that requires indentation. The indented lines following this statement are part of this block.
- **The Nested if False**: statement is indented further to indicate that it's nested inside the outer if block.
- The line **print("This line is also indented by one level.")** is at the same indentation level as the preceding print statement, so it's still part of the same block.
- The line **print("This line is at the outermost level.")** is not indented, indicating that it's outside any previous block.

Types of Indentation in Python

1. **Space Indentation**: Python conventionally uses spaces for indentation. A common practice is to use **four spaces for each level of indentation**. The use of spaces ensures consistent and visually clear code.

2. **Tab Indentation:** While spaces are recommended, tabs can also be used for indentation. However, **mixing tabs and spaces should be avoided**, as it can lead to inconsistent indentation.

Indentation is also used in various control structures:

- In if, elif, and else statements.
- In for and while loops.
- In function and class definitions.
- In context managers created with the with statement.

Proper indentation is essential to maintain code readability and to ensure that the intended program structure matches the actual structure. Incorrect or inconsistent indentation can lead to syntax errors or unexpected behavior in Python programs.

10

Statements

In Python, statements are individual lines of code that perform specific actions or operations. They are the building blocks of a program and are executed sequentially, one after the other. Each statement typically performs a specific task, such as assigning a value to a variable, calling a function, or controlling the flow of the program.

Python supports several types of statements, each serving a different purpose. Let's explore some of the most common types of statements along with examples:

Assignment Statements: Assignment statements are used to assign values to variables.

```
x = 5
name = "Alice"
```

Expression Statements: Expression statements consist of expressions that are evaluated, and their results are sometimes used or discarded.

```
y = x + 3      # An expression statement with an arithmetic expression
print(y)       # An expression statement with a function call
```

Print Statements: The print statement is used to output data to the console.

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

Conditional Statements: Conditional statements allow you to execute code based on conditions.

```
if x > 10:
   print("x is greater than 10")
else:
   print("x is not greater than 10")
```

Loop Statements: Loop statements are used to repeat a block of code multiple times.

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

Function Definition Statements: Function definition statements are used to create user-defined functions.

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

Import Statements: Import statements are used to bring external modules or libraries into your program.

```
import math
```

Break and Continue Statements: The break statement is used to exit a loop prematurely, while the continue statement skips the rest of the current iteration and moves to the next one.

```
for i in range(10):
   if i == 5:
      break
   print(i)
```

Return Statements: Return statements are used in functions to specify a value to be returned when the function is called.

```
def add(x, y):
   return x + y
```

Raise Statements: Raise statements are used to raise exceptions.

```
if x < 0:
   raise ValueError("x must be a positive number")
```

These are just a few examples of the many types of statements available in Python. Statements are the executable components of your code that define its behavior and functionality. Understanding how to use different types of statements is essential for effective programming in Python.

11

Variables

In Python, a variable is a symbolic name that refers to a value stored in the computer's memory. It allows you to store and manipulate data, making your code more flexible and dynamic. Variables enable you to give meaningful names to data values, making your code more readable and understandable.

Here's how you declare and use variables in Python, along with some examples:

Variable Declaration: In Python, you don't need to explicitly declare a variable's data type. You simply assign a value to a name using the assignment operator =.

```
# Variable assignment
age = 25
name = "John"
height = 5.11
is_student = True
```

Variable Names: Variable names must start with a letter (a-z, A-Z) or an underscore (_) and can be followed by letters, digits (0-9), or underscores. Variable names are case-sensitive.

```
# Valid variable names
first_name = "Alice"
_last_name = "Smith"
age_2023 = 30
```

Using Variables: You can use variables in expressions, assignments, and various operations.

```
# Using variables in expressions
birth_year = 1990
current_year = 2023
age = current_year - birth_year
print("Age:", age)
```

```
# Updating variables
count = 5
count = count + 1          # Incrementing count
print("Updated count:", count)
```

Dynamic Typing: Python is dynamically typed, meaning you can reassign variables to different types.

```
x = 10
print(x)                   # Output: 10

x = "Hello"
print(x)                   # Output: Hello
```

Multiple Assignment: You can assign multiple variables in a single line using commas.

```
a, b, c = 5, 10, 15
print("a:", a, "b:", b, "c:", c)
```

Swapping Values: Python allows you to easily swap the values of variables.

```
x = 5
y = 10
x, y = y, x                # Swapping values
print("x:", x, "y:", y)
```

Constants: While Python doesn't have true constants, it's a convention to use uppercase variable names to represent constant values.

```
PI = 3.14159
GRAVITY = 9.81
```

Remember that variables hold data in memory, and their values can change during the program's execution. Proper naming and usage of variables can enhance the clarity and maintainability of your code.

Rules for Naming Variables

When naming variables in Python, there are certain rules and conventions you should follow to ensure clarity, readability, and compatibility with the language's syntax. **Here are the key rules** for naming variables in Python:

Valid Characters: Variable names can consist of letters (both uppercase and lowercase), digits (0-9), and underscores (_). They must start with a letter or an underscore. Special characters, spaces, and punctuation marks are not allowed.

Case Sensitivity: Python variable names are case-sensitive. This means that myVariable, myvariable, and MYVARIABLE are all treated as different variables.

Reserved Keywords: You cannot use Python's reserved keywords (also known as keywords or reserved words) as variable names. These are words that have special meanings in Python and are used to define the language's structure and behavior. For example, you can't use words like if, while, for, True, False, def, etc., as variable names.

Meaningful Names: Choose descriptive and meaningful names for your variables. This improves code readability and helps others (and your future self) understand the purpose of the variable.

Snake Case: Use the snake_case naming convention for variables. This involves writing all lowercase letters and separating words with underscores. For example: user_name, total_amount, current_balance.

Avoid Single Letters: Except for some standard cases like loop iterators (i, j, k), avoid using single-letter variable names. It's usually better to give more descriptive names.

Avoid Ambiguous Names: Choose names that clearly represent the purpose of the variable. Avoid using names that could be ambiguous or misleading.

Avoid Leading Underscore: While starting a variable name with an underscore is allowed, it has a specific meaning in Python. A single leading underscore is often used to indicate a "**private**" variable, but it doesn't actually make the variable private or inaccessible. It's a convention to let others know that a variable or method is intended for internal use.

Multiple Words: If your variable name consists of multiple words, use underscores to separate them. This enhances readability. For example: student_name, item_count.

Length Consideration: Variable names can be of any length, but excessively long names might make the code harder to read. Aim for a balance between being descriptive and concise.

Constants: While Python doesn't have true constants, it's a convention to use uppercase letters and underscores to name variables that are intended to be treated as constants. For example: MAX_SIZE, PI, DEFAULT_COLOR.

Here are some examples of **valid variable names**:

```
age = 25
first_name = "John"
total_amount = 100.50
is_student = True
```

And here are some examples of invalid variable names:

```
1st_place = "Alice"      # Cannot start with a digit
my-variable = 10         # Cannot contain a hyphen
for = 5                  # Reserved keyword cannot be used
```

By following these rules and conventions, you'll create more readable and maintainable Python code.

12

Data Types

In programming, data types define the type of values that variables can hold. Each data type has specific characteristics and operations associated with it. Python is a dynamically typed language, meaning you don't need to declare the data type explicitly; the interpreter infers it based on the value assigned to the variable. Here are some of the common data types in Python:

Type		Meaning	Example
Numeric	**int**	Represents integers (whole numbers).	age = 25 count = -10
	float	Represents floating-point. numbers (decimal numbers)	temperature = 98.6 pi = 3.14159
	complex	Represents complex numbers in the form real + imaginary*j	2+3*j 4*j
Boolean		Represents a binary value, True or False.	is_student = True is_adult = False
String		Represents sequences of characters enclosed in single or double quotes.	name = "Alice" message = 'Hello, World!'
List		Represents an ordered collection of items, which can be of different data types.	numbers = [1, 2, 3, 4, 5] names = ["Alice", "Bob", "Charlie"]
Tuple		Similar to a list, but tuples are immutable (cannot be changed after creation).	coordinates = (3, 7) rgb_color = (255, 0, 0)
Set		Represents an unordered collection of unique elements.	unique_numbers = {1, 2, 3, 4, 5} unique_chars = {'a', 'b', 'c'}
Dictionary		Represents a collection of key-value pairs,	student = {"name": "Alice", "age": 20,

	where keys are unique.	"is_student": True} book = {"title": "Python Programming", "pages": 300}
None Type	Represents the absence of a value or a null value.	result = None

These are the fundamental data types in Python. It's important to understand the characteristics and appropriate use cases for each type, as it influences how you manipulate and process your data. Python also supports type conversion (casting) between different data types using built-in functions like `int()`, `float()`, `str()`, etc.

13

Operators

Operators in Python are symbols or special functions that perform operations on values or variables. They enable you to manipulate data and perform various calculations. Python supports a wide range of operators, categorized into different types:

Arithmetic Operators: These operators perform basic arithmetic operations.

```
a = 10
b = 3
addition = a + b      # 13
subtraction = a - b          # 7
multiplication = a * b       # 30
division = a / b             # 3.333...
floor_division = a // b      # 3 (floor division)
modulus = a % b              # 1 (remainder)
exponentiation = a ** b   # 1000 (a raised to the power of b)
```

Comparison Operators: These operators compare two values and return a boolean value (True or False).

```
x = 5
y = 7
is_equal = x == y            # False
not_equal = x != y           # True
greater_than = x > y         # False
less_than = x < y            # True
greater_equal = x >= y       # False
less_equal = x <= y          # True
```

Logical Operators: These operators perform logical operations on boolean values.

```
p = True
q = False
```

```
logical_and = p and q      # False
logical_or = p or q        # True
logical_not = not p        # False
```

Assignment Operators: These operators are used to assign values to variables.

```
x = 10
y = 5
x += y        # Equivalent to x = x + y
x -= y        # Equivalent to x = x - y
x *= y        # Equivalent to x = x * y
x /= y        # Equivalent to x = x / y
x //= y       # Equivalent to x = x // y
x %= y        # Equivalent to x = x % y
x **= y       # Equivalent to x = x ** y
```

Bitwise Operators: These operators perform bit-level operations on integers.

```
a = 5
b = 3
bitwise_and = a & b       # 1 (bitwise AND)
bitwise_or = a | b        # 7 (bitwise OR)
bitwise_xor = a ^ b       # 6 (bitwise XOR)
bitwise_not = ~a          # -6 (bitwise NOT)
left_shift = a << 1       # 10 (left shift)
right_shift = a >> 1      # 2 (right shift)
```

Membership Operators: These operators test if a value is a member of a sequence (like strings, lists, or sets).

```
colors = ['red', 'green', 'blue']
is_red_present = 'red' in colors              # True
is_yellow_present = 'yellow' in colors        # False
```

Identity Operators: These operators compare the memory addresses of two objects.

```
x = [1, 2, 3]
y = x
is_same_object = x is y                 # True (same memory address)
is_different_object = x is not y        # False (same memory address)
```

Unary Operators: These operators perform operations on a single operand.

```
a = 5
positive = +a        # 5 (unary plus)
negative = -a        # -5 (unary minus)
```

These are the primary types of operators in Python. Understanding how to use them effectively is crucial for performing various tasks in programming.

14

Pemdas Rule

PEMDAS is an acronym that stands for Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right). It's a rule used to determine the order of operations when evaluating mathematical expressions. This rule ensures that expressions are evaluated consistently and accurately.

Let's break down each part of the PEMDAS rule with examples:

Parentheses: Perform operations within parentheses first.

```
result = (3 + 5) * 2
# Inside parentheses: 3 + 5 = 8
# Multiply by 2: 8 * 2 = 16
# Result: 16
```

Exponents: Evaluate exponential operations.

```
result = 2 ** 3 + 1
# 2 raised to the power of 3: 2 ** 3 = 8
# Add 1: 8 + 1 = 9
# Result: 9
```

Multiplication and Division: Evaluate multiplication and division from left to right.

```
result = 10 / 2 * 3
# Divide 10 by 2: 10 / 2 = 5
# Multiply by 3: 5 * 3 = 15
# Result: 15
```

Addition and Subtraction: Evaluate addition and subtraction from left to right.

```
result = 8 - 3 + 2
# Subtract 3 from 8: 8 - 3 = 5
# Add 2: 5 + 2 = 7
# Result: 7
```

By following the PEMDAS rule, you ensure that mathematical expressions are evaluated correctly. Keep in mind that if there are nested parentheses, you'll

start with the innermost set and work your way outward, applying the order of operations at each step. This helps avoid ambiguity and ensures that calculations are performed consistently.

Example: Let's consider the expression (8 - 3) ** 2 * 4 / 2 + 10 / 2 and walk through its evaluation step by step using the PEMDAS rule.

Expression: (8 - 3) ** 2 * 4 / 2 + 10 / 2

Parentheses: We start by evaluating the operations within parentheses:

(8 - 3) ** 2 * 4 / 2 + 10 / 2
= 5 ** 2 * 4 / 2 + 10 / 2
= 25 * 4 / 2 + 10 / 2

Exponents: Now we evaluate the exponentiation operation:

25 * 4 / 2 + 10 / 2
= 100 / 2 + 10 / 2

Multiplication and Division: We perform multiplication and division from left to right:

100 / 2 + 10 / 2
= 50 + 10 / 2
= 50 + 5

Addition and Subtraction: Finally, we perform addition:

50 + 5
= 55

So, the result of the expression (8 - 3) ** 2 * 4 / 2 + 10 / 2 is 55.

This example demonstrates how the PEMDAS rule ensures that we follow a specific order of operations when evaluating expressions. It's important to understand this rule to accurately compute the results of mathematical expressions in a consistent manner.

15

Operator Precedence

Operator precedence, also known as operator priority, establishes the order of evaluation for operators in expressions with multiple operators. It ensures consistent and accurate mathematical evaluation, preventing ambiguity. In Python, like many languages, operators are prioritized based on a hierarchy. Those with higher precedence are evaluated first, and when precedence is equal, associativity determines the order.

The Python operator precedence table (given below) categorizes operators from top to bottom, indicating their tightness of binding, with operators in the same row having equal precedence.

This system ensures proper calculation, as seen in expressions like 3 + 5 * 2, where multiplication precedes addition, yielding 13.

	Operator	**Description**	**Associativity**
1.	()	Parentheses (grouping)	No associativity, as they are used for grouping and have no inherent direction.
2.	[]	Brackets (list indexing)	Left to Right
3.	{}	Curly braces (dictionary, set)	No associativity, as they are used for dictionary and set literals.
4.	.	Dot (attribute access)	Left to Right
5.	**	Exponentiation	Right to Left
6.	+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT	Right to Left
7.	*, /, %	Multiplication, Division, Modulo	Left to Right
8.	+, -	Addition, Subtraction	Left to Right
9.	<<, >>	Bitwise left shift, Bitwise right shift	Left to Right
10.	&	Bitwise AND	Left to Right
11.	^	Bitwise XOR	Left to Right
12.	|	Bitwise OR	Left to Right

13.	in, not in, is, is not	Comparisons, Identity	No associativity, as these are binary operators that compare two values.
14.	<, <=, >, >=	Comparisons	
15.	==, !=	Equality, Inequality	Left to Right
16.	not	Logical NOT	Right to Left
17.	and	Logical AND	Left to Right
18.	or	Logical OR	Left to Right
19.	=, +=, -=, *=, /=, //=, %=, **=, &=, |=, ^=, <<=, >>=	Assignment	Right to Left

Example: Consider the expression given below

```
x = [1, 2, 3]
y = {4, 5, 6}
z = 7
result = (z ** 2 + (-(x[1] * 3) // y.pop()) & 9) ^ (8 | z)
```

Evaluation of Expression takes place as follows

1. **Parentheses**
   x[1] is evaluated: 2
   x[1] * 3 is evaluated: 6
   y.pop() removes and returns an element from set y: Let's say it's 4
   Now the expression becomes: (z ** 2 + (-6 // 4) & 9) ^ (8 | z)

2. **Exponents**
   z ** 2 is evaluated: 49
   Now the expression becomes: (49 + (-6 // 4) & 9) ^ (8 | z)

3. **Unary Minus, Division, and Floor Division**
   -6 // 4 is evaluated: -2
   Now the expression becomes: (49 + (-2) & 9) ^ (8 | z)

4. **Bitwise AND**
   -2 & 9 is evaluated: 8
   Now the expression becomes: (49 + 8) ^ (8 | z)

5. **Bitwise OR**
   8 | z is evaluated: 15
   Now the expression becomes: (49 + 8) ^ 15

6. **Addition**
   49 + 8 is evaluated: 57
   Now the expression becomes: 57 ^ 15

7. **Bitwise XOR**
   57 ^ 15 is evaluated: 50

8. **Assignment:**
   The result 50 is assigned to the variable result.

So, after evaluating the entire expression, the value of the variable result becomes 50. This process demonstrates how the Python operator precedence table guides the order of operations to compute the final result of the expression.

16

Control Statements

Control statements in programming are used to determine the flow of execution of a program. They allow you to make decisions, repeat actions, and control the order in which different parts of your code are executed. There are three main types of control statements: **conditional statements, looping statements, and branching statements**.

Here's a brief explanation of each type along with their syntax in a typical programming language like Python:

1. **Conditional Statements**: Conditional statements are used to make decisions in your code. They allow you to execute different blocks of code based on whether a certain condition is **true or false**. Here are examples of conditional statements using **if, elif, and else:**

   If Statement: The if statement is used to execute a block of code if a certain condition is true.

   Syntax

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

   Example

   ```
   age = 20
   if age >= 18:
      print("You are an adult.")
   ```

   If-Else Statement: The if-else statement is used to execute one block of code if a condition is true, and another block if the condition is false.

   Syntax

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

Example

```
age = 18
if age >= 18:
   print("You are an adult.")
else:
   print("You are not yet an adult.")
```

elif statement: The elif statement, short for "else if," is used in conjunction with an if statement to provide an additional condition to check when the initial if condition is false. It allows you to test multiple conditions in sequence and execute different blocks of code based on which condition is true. Here's the syntax for using elif in a typical programming language like Python:

```
if  condition1:
   # Code to be executed if condition1 is true
elif  condition2:
   # Code to be executed if condition2 is true
elif  condition3:
   # Code to be executed if condition3 is true
# ... (you can have more elif blocks)
else:
   # Code to be executed if none of the above conditions are true
```

The elif statement allows you to create a chain of conditions that are tested one after the other. If the first if condition is true, its corresponding code block is executed. If the first if condition is false, the program moves on to the next elif condition. This process continues until either a condition is true or the final else block (if present) is executed.

Here's a simple example:

```
score = 85
if score >= 90:
   grade = "A"
elif score >= 80:
   grade = "B"
elif score >= 70:
   grade = "C"
else:
   grade = "D"
```

```
print("Your grade:", grade)
```

2. **Looping Statements:** Looping statements allow you to execute a block of code repeatedly as long as a certain condition remains true.

   Syntax (in Python)

```
while condition:
   # Code to be executed repeatedly as long as the condition is true
```

   Example

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

3. **Branching Statements:** Branching statements provide a way to alter the flow of execution by transferring control to a different part of the program.

   Break: The break statement is used to exit the current loop prematurely, regardless of whether the loop's condition is still true.

   Syntax (in Python)

```
while condition:
   if some_condition:
      break
```

   Example

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

for number in numbers:
   if number == 5:
      break            # Exit the loop when number is 5
   print(number)

# Output: 1 2 3 4
```

4. **Continue:** The continue statement is used to skip the current iteration of a loop and continue with the next iteration.

   Syntax (in Python)

```
for item in sequence:
```

```
if some_condition:
    continue
```

Example

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

for number in numbers:
    if number % 2 == 0:
        continue                # Skip even numbers.
    print(number)

# Output: 1 3 5 7 9
```

5. **Return:** In functions, the return statement is used to exit the function and optionally return a value to the caller.

Syntax (in Python)

```
def  my_function():
    # Code
    return some_value
```

Example

```
def  add_numbers(a, b):
    result = a + b
    return result

sum_result = add_numbers(5, 3)
print("Sum:", sum_result)

# Output: Sum: 8
```

These control statements are fundamental building blocks in programming that allow you to create dynamic and responsive applications by controlling the flow of execution based on different conditions and requirements. The syntax provided here is in Python, but similar concepts exist in most programming languages with some variations in syntax.

17

Pass

In programming, the **pass** statement is used as a placeholder for future code. It doesn't perform any action and is often used when a statement is syntactically required but you don't want to execute any code at that point. It's commonly used during development when you're creating the structure of your code and plan to fill in the details later.

Here's an example to illustrate the use of the pass statement:

```
def process_data(data):
    if  len(data) > 10:
        # TODO: Implement code to process large data
        pass
    else:
        # TODO: Implement code to process small data
        pass

# The implementation of processing large and small data is not ready yet,
# but the code structure is in place using the pass statement.
```

In this example, the **process_data** function is intended to process data based on its length. If the data is larger than 10 elements, the plan is to process it in a certain way (which hasn't been implemented yet). If the data is smaller or equal to 10 elements, there's another intended processing method (also not yet implemented).

By using the pass statement in these blocks, you're indicating that these portions of code are intentionally left blank and will be filled in later. This helps you maintain the overall structure of your code while you're in the process of development or when you're writing pseudocode to plan out your program's logic.

18

Functions

In programming, a function is a self-contained block of code that performs a specific task or a set of related tasks. Functions allow you to break down your code into smaller, reusable components, making it easier to manage, debug, and maintain your codebase. In Python, there are several types of functions:

Built-in Functions: Python comes with a variety of built-in functions that are readily available for use without needing to define them yourself. These functions cover a wide range of tasks, such as converting data types, performing mathematical operations, and working with collections.

Example

```
num_list = [5, 2, 8, 1, 3]
max_value = max(num_list)
print("Maximum value:", max_value)
```

User-Defined Functions: User-defined functions are functions that you create yourself to perform specific tasks. They allow you to encapsulate a sequence of statements into a single unit that can be called and reused.

Syntax

```
def function_name(parameters):
   # Function body
   return result                          # Optional
```

Example

```
def greet(name):                  # Function Definition
   return "Hello, " + name

message = greet("Alice")        # Function Call
print(message)
```

Lambda Functions (Anonymous Functions): Lambda functions, also known as anonymous functions, are small, one-line functions defined without a name. They are typically used for simple operations where you don't need a full function definition.

Syntax

```
lambda arguments: expression
```

Example

```
square = lambda x: x * x
result = square(5)
print(result)
```

Recursive Functions: Recursive functions are functions that call themselves in their own definition. They are useful for solving problems that can be broken down into smaller, similar sub-problems.

Example (calculating factorial using recursion):

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

result = factorial(5)
print(result)
```

Higher-Order Functions: Higher-order functions are functions that can take other functions as arguments or return functions as results. They enable powerful functional programming paradigms.
Example (using the map function to apply a function to all elements of a list):

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x * x, numbers))
print(squared)
```

These are some of the main types of functions in Python. Functions play a crucial role in organizing and modularizing your code, making it more readable and maintainable.

19

Types of User Defined Functions

User-defined functions are functions that you create in your code to perform specific tasks. They allow you to encapsulate a sequence of statements into a single unit that can be called and reused. User-defined functions enhance code readability, modularity, and reusability.

Based on return values and arguments, there are several types of user-defined functions:

Function with No Arguments and No Return Value

These functions don't take any input arguments and don't return any value.

Example

```
def greet():
   print("Hello!")

greet()          # Calling the function
```

Function with Arguments and No Return Value:

These functions accept input arguments (parameters) but don't return any value.

Example

```
def add_numbers(a, b):
   sum_result = a + b
   print("Sum:", sum_result)

add_numbers(5, 3)                # Calling the function with arguments
```

Function with Arguments and Return Value

These functions take input arguments and return a computed value.

Example

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

```
    product = a * b
    return product

result = multiply(4, 6)          # Calling the function and storing the
result
print("Product:", result)
```

Function with Default Arguments

Default arguments are used when a value is not provided for a certain parameter during function call.

Example

```
def power(base, exponent=2):
    result = base ** exponent
    return result

print(power(3))          # Using default exponent (2)
print(power(2, 4))              # Using provided exponent (4)
```

Function with Variable Number of Arguments (Variadic Functions)

These functions accept a variable number of arguments using the *args syntax.

Example

```
def average(*numbers):
    total = sum(numbers)
    count = len(numbers)
    return total / count

print(average(5, 10, 15))    # Average of 5, 10, and 15
print(average(2, 6, 8, 10))  # Average of 2, 6, 8, and 10
```

Function with Keyword Arguments

Keyword arguments are used to specify parameter values by name, enhancing readability.

Example

```
def person_info(name, age):
    print("Name:", name)
    print("Age:", age)
```

```
person_info(age=25, name="Alice")
```

Function with Multiple Return Values

Python allows functions to return multiple values as a tuple. Even though the syntax involves returning a single tuple, you can unpack the values into separate variables when you receive the return value.
Here's how you can define and use a function that returns multiple values:

```
def calculate_statistics(data):
    mean = sum(data) / len(data)
    variance = sum((x - mean) ** 2 for x in data) / len(data)
    min_value = min(data)
    max_value = max(data)
    return mean, variance, min_value, max_value

data = [5, 8, 10, 12, 15]
mean_value, variance_value, min_value, max_value =
calculate_statistics(data)

print("Mean:", mean_value)
print("Variance:", variance_value)
print("Minimum Value:", min_value)
print("Maximum Value:", max_value)
```

In this example, the calculate_statistics function computes the mean, variance, minimum, and maximum values of a given dataset. The function returns a tuple containing these four values. When calling the function, the returned tuple is unpacked into separate variables, allowing you to access each value individually.

20

Keyword Arguments

Keyword arguments, also known as named arguments, allow you to pass arguments to a function using their parameter names. This enhances code readability by making it clear which value corresponds to which parameter, especially when a function has multiple parameters.
Here's how you can use keyword arguments in Python functions:

```
def person_info(name, age):
    print("Name:", name)
    print("Age:", age)
person_info(name="Alice", age=25)  # Using keyword arguments
```

In this example, the person_info function takes two parameters: name and age. When calling the function, you provide the values using their parameter names, separated by **equals signs (=).** This makes the code more self-explanatory and less error-prone, especially when the function has multiple parameters.

Keyword arguments also allow you to provide arguments **in any order**, as long as you specify their names:

```
person_info(age=30, name="Bob")  # Order of keyword arguments doesn't matter
```

Using keyword arguments is particularly helpful when a function has default values for some parameters:

```
def greet(name, greeting="Hello"):
    print(greeting, name)

greet(name="Alice")        # Using default greeting ("Hello")
greet(name="Bob", greeting="Hi")  #Providing a custom greeting ("Hi")
```

In this example, the greet function has a default greeting of "Hello." When calling the function, you can explicitly provide a value for the greeting parameter using a keyword argument.

Using keyword arguments makes your code more readable, self-explanatory, and flexible when working with functions that have multiple parameters or default values.

21

Local and Global Scope

In Python, "scope" refers to the region of the code where a particular variable or name can be accessed. Python has two main types of variable scopes: local scope and global scope.

Local Scope: Variables defined within a function have local scope. They can only be accessed within the function where they are defined. Once the function completes execution, the local variables are destroyed and cannot be accessed from outside the function.

Example

```
def my_function():
    local_var = 10  # This is a local variable
    print(local_var)

my_function()
# Accessing local_var here would result in an error
```

Global Scope: Variables defined outside of any function have global scope. They can be accessed from anywhere in the code, both inside and outside functions.

Example

```
global_var = 20  # This is a global variable

def another_function():
    print(global_var)  # Accessing global_var here

another_function()
print(global_var)  # Accessing global_var here as well
```

It's important to note that when you assign a value to a variable within a function, Python considers it a local variable by default, even if a variable with the same name exists in the global scope.

Example illustrating global and local scope interactions

```
x = 5  # Global variable
def my_function():
   x = 10  # This creates a new local variable x within the function
   print(x)  # Prints the local x (10)

my_function()
print(x)  # Prints the global x (5)
```

To modify a global variable from within a function, you need to explicitly declare it as a global variable using the global keyword.

Example modifying a global variable within a function

```
x = 5  # Global variable

def modify_global():
   global x  # Declare x as a global variable
   x = 15
modify_global()
print(x)  # Prints the modified global x (15)
```

It's generally a good practice to avoid using global variables as much as possible, as they can make code harder to understand and maintain. Instead, prefer passing values as function arguments and returning values from functions to maintain better control over variable scope.

22

Lambda Function

Lambda functions, also known as **anonymous functions**, are a concise way to create small, **one-line functions** in Python. They are often used when you need a simple function for a short period of time and don't want to define a regular named function using the def keyword. Lambda functions are defined using the **lambda** keyword, followed by the input parameters and the expression to be evaluated.

The general syntax of a lambda function is:

lambda arguments: expression

Here are some examples to illustrate the concept of lambda functions:

Basic Example: A lambda function to calculate the square of a number:

```
square = lambda  x: x ** 2
result = square(4)          # Result will be 16
```

Sorting with Lambda: Lambda functions are often used as the key parameter in sorting functions to customize the sorting criteria:

```
students = [("Alice", 25), ("Bob", 22), ("Charlie", 28)]
students.sort(key=lambda student: student[1])      # Sort based on age
```

Using Lambda with map(): Lambda functions are frequently used with the map() function to apply a function to each element of an iterable:

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x ** 2, numbers)  # Applies lambda to each number
```

Filtering with Lambda: Lambda functions can be used with the filter() function to filter elements based on a condition:

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)  # Filters even numbers
```

Using Lambda in Key Functions: Lambda functions can be used as key functions for more complex operations, like getting the max item based on a specific property:

```
data = [("apple", 3), ("banana", 5), ("cherry", 1)]
max_item = max(data, key=lambda item: item[1])  # Gets item with max count
```

Lambda functions are useful for quick operations, but keep in mind that they're limited to single expressions and shouldn't be used for complex logic. In cases where you need more complex logic or reusability, it's better to define regular functions using the def keyword.

23

Mapping

Mapping is a fundamental concept in programming that involves applying a function to every element of a collection (like a list) and getting a new collection with the results. Python provides several ways to perform mapping, such as using loops, list comprehensions, and the built-in map() function. Mapping is particularly useful for transforming data and performing operations on each element of a collection.

Here are examples of various mapping techniques in Python:

Using a Loop

Mapping using a loop involves iterating through each element of a collection, applying a function, and collecting the results in a new list.

```
numbers = [1, 2, 3, 4, 5]
squared = []
for num in numbers:
   squared.append(num ** 2)
# squared is now [1, 4, 9, 16, 25]
```

Using List Comprehensions

List comprehension provides a concise way to create new lists by applying an expression to each element of an existing list.

```
numbers = [1, 2, 3, 4, 5]
squared = [num ** 2 for num in numbers]
# squared is now [1, 4, 9, 16, 25]
```

Using the map() Function

The built-in map() function applies a given function to each item of an iterable (e.g., a list) and returns an iterator containing the results.

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

```
squared = map(lambda x: x ** 2, numbers)
# squared is a map object, so you can convert it to a list
squared_list = list(squared)  # [1, 4, 9, 16, 25]
```

Mapping with Multiple Iterables

The map() function can accept multiple iterables as arguments. The function provided should also accept the same number of arguments.

```
numbers = [1, 2, 3]
multipliers = [10, 20, 30]
results = map(lambda x, y: x * y, numbers, multipliers)
# results is an iterator: [10, 40, 90]
```

Mapping with Built-in Functions

Python's built-in functions, like **len()** and **str()**, can also be used with mapping techniques.

```
words = ["apple", "banana", "cherry"]
lengths = map(len, words)
# lengths is an iterator: [5, 6, 6]
```

Mapping is a versatile technique that helps you efficiently transform data and perform operations on entire collections, without needing explicit loops. Depending on your specific use case and preference, you can choose the mapping technique that suits your needs best.

24

Filter

Filtering is a concept in programming that involves selecting elements from a collection (like a list) that meet a certain condition. Python provides various ways to perform filtering, such as using loops, list comprehensions, and the built-in filter() function. Filtering is useful for extracting specific elements from a collection based on some criteria.

Here are examples of filtering techniques in Python:

Using a Loop

Filtering using a loop involves iterating through each element of a collection, checking a condition, and adding elements that satisfy the condition to a new list.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
even_numbers = []
for num in numbers:
   if num % 2 == 0:
      even_numbers.append(num)
# even_numbers is now [2, 4, 6, 8]
```

Using List Comprehensions: List comprehensions provide a concise way to create new lists by applying a condition to each element of an existing list.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
even_numbers = [num for num in numbers if num % 2 == 0]
# even_numbers is now [2, 4, 6, 8]
```

Using the filter() Function: The built-in filter() function filters an iterable (e.g., a list) based on a given function and returns an iterator containing the elements that satisfy the condition.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
# even_numbers is an iterator, so you can convert it to a list
```

```
even_numbers_list = list(even_numbers)          # [2, 4, 6, 8]
```

Filtering with Built-in Functions: Python's built-in functions, like str.isnumeric() or str.startswith(), can also be used with filtering techniques.

```
words = ["apple", "banana", "123", "cherry"]
numeric_words = filter(str.isnumeric, words)
# numeric_words is an iterator: ["123"]
```

Filtering with Multiple Conditions: You can use logical operators (and, or, not) to combine multiple conditions for filtering.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
filtered_numbers = filter(lambda x: x % 2 == 0 or x % 3 == 0, numbers)
# filtered_numbers is an iterator: [2, 3, 4, 6, 8, 9]
```

Filtering allows you to extract specific elements from a collection based on criteria, making it a powerful tool for data manipulation and selection. Choose the filtering technique that best fits your specific use case and coding style.

25

Exception Handling

Exception handling is a programming concept that allows you to gracefully handle and manage errors or exceptions that may occur during the execution of your code. Instead of letting errors crash your program, you can use exception handling to catch these errors, provide meaningful feedback to users, and potentially recover from the error. In Python, exceptions are raised when an error occurs, and you can use try, except, else, and finally blocks to manage them.

Here's how exception handling works, along with examples

Basic Exception Handling

Use the try and except blocks to handle exceptions. If an exception occurs within the try block, the code in the corresponding except block is executed.

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

Handling Multiple Exceptions

You can handle multiple exceptions using multiple except blocks.

```
try:
    value = int(input("Enter a number: "))
    result = 10 / value
    print("Result:", result)
except (ZeroDivisionError, ValueError):
    print("Error: Invalid input or division by zero.")
```

Using else Block

The else block is executed when no exceptions occur in the try block

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Result:", result)
```

Using Finally Block

The finally block is executed no matter what, whether an exception occurred or not. It's often used for cleanup operations.

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    if file:
        file.close()
```

Catching All Exceptions

You can use a generic except block to catch all exceptions, but this should be used carefully as it might hide unexpected errors.

```
try:
    # code that might raise exceptions
except Exception as e:
    print("An error occurred:", e)
```

Raising Exceptions

You can use the raise statement to explicitly raise exceptions in your code

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

```
try:
    value = int(input("Enter a positive number: "))
    check_positive(value)
except ValueError as ve:
    print("Error:", ve)
```

Exception handling is crucial for writing robust and reliable code. By anticipating and handling errors effectively, you can provide a better user experience and make your programs more stable.

26

Built in Functions

Built-in functions, also known as standard library functions, are pre-defined functions that are provided by programming languages or libraries. These functions serve specific purposes and are readily available for use without needing to be explicitly defined by the programmer. They can help perform common tasks, manipulate data, and interact with the system.

In Python, you can use the dir() function to get a list of names in the current module's namespace. Since many built-in functions are available by default in Python, you can use dir(__builtins__) to get a list of built-in function names. Here's the command:

print(dir(__builtins__))
#Output:
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod',

'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'execfile', 'filter', 'float', 'format', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'runfile', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

Here are some examples of built-in functions in Python

Function	Meaning	Example Usage
abs()	Returns the absolute value of a number.	abs(-5) returns 5
len()	Returns the length of a sequence (string, list, etc.).	len("Hello") returns 5
type()	Returns the type of an object.	type(5) returns <class 'int'>
print()	Displays output to the console.	print("Hello, world!")
input()	Takes user input from the console.	name = input("Enter your name: ")
str()	Converts an object to a string.	str(42) returns '42'
int()	Converts a string or number to an integer.	int("5") returns 5
float()	Converts a string or number to a floating-point num.	float("3.14") returns 3.14
list()	Converts an iterable to a list.	list("hello") returns ['h', 'e', 'l', 'l', 'o']
dict()	Creates a dictionary from a sequence of key-value.	dict([('a', 1), ('b', 2)])
max()	Returns the maximum value from a sequence.	max([5, 2, 8]) returns 8
min()	Returns the minimum value from a sequence.	min([5, 2, 8]) returns 2
sum()	Returns the sum of elements in a sequence.	sum([1, 2, 3]) returns 6
sorted()	Returns a sorted list from a sequence.	sorted([3, 1, 4]) returns [1, 3, 4]
range()	Generates a sequence of numbers within a range.	list(range(1, 5)) returns [1, 2, 3, 4]
round()	Rounds a floating-point number to a specified precision.	round(3.14159, 2) returns 3.14
zip()	Combines multiple iterables into tuples.	list(zip([1, 2, 3], ['a', 'b', 'c']))

enumerate()	Returns an enumerate object with index-value pairs.	list(enumerate(['a', 'b', 'c']))
chr()	Returns a character from an ASCII value.	chr(65) returns 'A'
ord()	Returns the ASCII value of a character.	ord('A') returns 65
abs()	Returns the absolute value of a number.	abs(-5) returns 5
len()	Returns the length of a sequence.	len("Hello") returns 5
all()	Returns True if all elements in an iterable are true.	all([True, True, False]) returns False
any()	Returns True if any element in an iterable is true.	any([True, False, False]) returns True
pow()	Raises a number to a power.	pow(2, 3) returns 8

27

List

In Python, a list is a built-in data structure that allows you to store and organize a collection of values. Lists are one of the most commonly used data types in Python and provide a flexible and versatile way to work with collections of items.

A list in Python

- **Is Ordered:** The elements in a list are ordered, meaning they have a specific position or index within the list. This order is maintained unless you explicitly change it.
- **Is Mutable:** Lists are mutable, which means you can modify their contents after they are created. You can add, remove, or change elements within a list.
- **Can Contain Different Data Types:** Lists can hold elements of various data types, including integers, floats, strings, and even other lists.

Lists in Python are incredibly useful for storing and manipulating collections of data. They are used in a wide range of applications, from basic data storage to more complex data processing tasks. Lists also support various operations such as slicing, sorting, counting elements, and more.

List Operations	Code Example	Description
Creating Lists		
Create	fruits = ["apple", "banana", "orange"]	Create a list with elements
Empty List	empty_list = []	Create an empty list
Accessing Elements		
Indexing	fruits[0]	Access element at index 0
Negative Indexing	fruits[-1]	Access last element using negative index
Modifying Lists		
Appending	fruits.append("grape")	Add element to the end of the list
Extending	fruits.extend(["peach",	Add multiple elements

	"pear"])	from another list
Inserting	fruits.insert(1, "kiwi")	Insert element at a specific index
Removing	fruits.remove("banana")	Remove element by value
Popping	popped = fruits.pop(1)	Remove and return element at index 1
List Operations		
Length	len(fruits)	Get the number of elements in the list
Sorting	fruits.sort()	Sort the list in ascending order
Reversing	fruits.reverse()	Reverse the order of elements
Counting	count = fruits.count("apple")	Count occurrences of an element
Index	index = fruits.index("orange")	Get the index of the first occurrence
Copying		
Shallow Copy	new_list = old_list.copy()	Create a shallow copy of the list
Deep Copy	import copy; new_list = copy.deepcopy(old_list)	Create a deep copy

28

List Slicing

List slicing is a way to extract a portion of a list in Python by specifying a range of indices. It allows you to create a new list containing elements from the original list within that specified range.

The syntax for list slicing is **list[start: end: step]**,

where

- start is the index of the first element you want to include (**inclusive**).
- end is the index of the first element you want to exclude (**exclusive**).
- step is the interval between elements to be included. It's optional and defaults to 1.

Here's an example to illustrate list slicing:

```
original_list = [10, 20, 30, 40, 50, 60, 70, 80]

# Positive Slicing
sliced_positive = original_list[2:5]  # [30, 40, 50]

# Negative Slicing
sliced_negative = original_list[-4:-1]  # [50, 60, 70]

# Slicing with Steps
sliced_step = original_list[1:7:2]  # [20, 40, 60]

# Omitted Start/End
sliced_omitted = original_list[:4]  # [10, 20, 30, 40]

# Full Slice
full_slice = original_list[:]  # [10, 20, 30, 40, 50, 60, 70, 80]
```

In this example, **sliced_positive** contains elements with indices 2, 3, and 4 from the original list, **sliced_negative** contains elements with indices -4, -3, and -2 (which correspond to the same elements as in sliced_positive), **sliced_step**

includes every second element starting from index1, **sliced_omitted** includes elements from the start up to index3, and **full_slice** is a copy of the original list.

Example 1: Here is a list of all possible positive and negative slice notations for the given list **x = [10, 20, 30, 40, 50, 60, 70, 80]**

Positive Slices of x = [10, 20, 30, 40, 50, 60, 70, 80]

x[0:1] -> [10]
x[0:2] -> [10, 20]
x[0:3] -> [10, 20, 30]
x[0:4] -> [10, 20, 30, 40]
x[0:5] -> [10, 20, 30, 40, 50]
x[0:6] -> [10, 20, 30, 40, 50, 60]
x[0:7] -> [10, 20, 30, 40, 50, 60, 70]
x[0:8] -> [10, 20, 30, 40, 50, 60, 70, 80]
x[:1] -> [10]
x[:2] -> [10, 20]
x[:3] -> [10, 20, 30]
x[:4] -> [10, 20, 30, 40]
x[:5] -> [10, 20, 30, 40, 50]
x**[:6]** -> [10, 20, 30, 40, 50, 60]
x[:7] -> [10, 20, 30, 40, 50, 60, 70]
x[:8] -> [10, 20, 30, 40, 50, 60, 70, 80]
x[1:] -> [20, 30, 40, 50, 60, 70, 80]
x[2:] -> [30, 40, 50, 60, 70, 80]
x[3:] -> [40, 50, 60, 70, 80]
x[4:] -> [50, 60, 70, 80]
x[5:] -> [60, 70, 80]
x[6:] -> [70, 80]
x[7:] -> [80]

Negative Slices of x = [10, 20, 30, 40, 50, 60, 70, 80]

x[-1:] -> [80]
x[-2:] -> [70, 80]
x[-3:] -> [60, 70, 80]
x[-4:] -> [50, 60, 70, 80]
x[-5:] -> [40, 50, 60, 70, 80]
x[-6:] -> [30, 40, 50, 60, 70, 80]
x[-7:] -> [20, 30, 40, 50, 60, 70, 80]
x[-8:] -> [10, 20, 30, 40, 50, 60, 70, 80]
x[:-1] -> [10, 20, 30, 40, 50, 60, 70]

x[:-2] -> [10, 20, 30, 40, 50, 60]
x[:-3] -> [10, 20, 30, 40, 50]
x[:-4] -> [10, 20, 30, 40]
x[:-5] -> [10, 20, 30]
x[:-6] -> [10, 20]
x[:-7] -> [10]

Empty Slices of x = [10, 20, 30, 40, 50, 60, 70, 80]

x[8:8] -> []
x[2:2] -> []
x[4:4] -> []
x[-3:-3] -> []

Omitted Start/End of x = [10, 20, 30, 40, 50, 60, 70, 80]

x[:1] -> [10]
x[:5] -> [10, 20, 30, 40, 50]
x[:-1] -> [10, 20, 30, 40, 50, 60, 70]
x[2:] -> [30, 40, 50, 60, 70, 80]
x[4:] -> [50, 60, 70, 80]
x[7:] -> [80]

Full Slices of x = [10, 20, 30, 40, 50, 60, 70, 80]

x[:] -> [10, 20, 30, 40, 50, 60, 70, 80]
x[::1] -> [10, 20, 30, 40, 50, 60, 70, 80]
x[::-1] -> [80, 70, 60, 50, 40, 30, 20, 10]
x[::2] -> [10, 30, 50, 70]
x[1::2] -> [20, 40, 60, 80]
x[1:7:2] -> [20, 40, 60]
x[-1::-1] -> [80, 70, 60, 50, 40, 30, 20, 10]

Step Slices of x = [10, 20, 30, 40, 50, 60, 70, 80]

x[::1] -> [10, 20, 30, 40, 50, 60, 70, 80]
x[::-1] -> [80, 70, 60, 50, 40, 30, 20, 10]
x[::2] -> [10, 30, 50, 70]
x[1::2] -> [20, 40, 60, 80]
x[1:7:2] -> [20, 40, 60]

Combination of Slicing Parameters of x = [10, 20, 30, 40, 50, 60, 70, 80]

```
x[2:5]   -> [30, 40, 50]
x[3:6]   -> [40, 50, 60]
x[1:7:2] -> [20, 40, 60]
x[-3:-1] -> [50, 60]
x[-6:-2] -> [30, 40, 50, 60]
x[2:7:3] -> [30, 60]
```

Negative Step Slices

```
x[::-1]      -> [80, 70, 60, 50, 40, 30, 20, 10]
x[::-2]      -> [80, 60, 40, 20]
x[-1:-9:-1]  -> [80, 70, 60, 50, 40, 30, 20, 10]
x[-1:-9:-2]  -> [80, 60, 40, 20]
x[-2:-9:-2]  -> [70, 50, 30, 10]
x[-3:-9:-2]  -> [60, 40, 20]
x[-4:-9:-2]  -> [50, 30, 10]
```

Omitted Step of x = [10, 20, 30, 40, 50, 60, 70, 80]

```
x[::]        -> [10, 20, 30, 40, 50, 60, 70, 80]
x[1::]       -> [20, 30, 40, 50, 60, 70, 80]
x[:7:]       -> [10, 20, 30, 40, 50, 60, 70]
x[:-1:]      -> [10, 20, 30, 40, 50, 60, 70]
```

Combination of All Parameters of x = [10, 20, 30, 40, 50, 60, 70, 80]

```
x[1:7:2]     -> [20, 40, 60]
x[-2:-8:-2]  -> [70, 50, 30]
x[4::-2]     -> [50, 30, 10]
```

Combination of All Parameters with Full Slices of x = [10, 20, 30, 40, 50, 60, 70, 80]

```
x[:]         -> [10, 20, 30, 40, 50, 60, 70, 80]
x[::]        -> [10, 20, 30, 40, 50, 60, 70, 80]
x[:8]        -> [10, 20, 30, 40, 50, 60, 70, 80]
x[1::]       -> [20, 30, 40, 50, 60, 70, 80]
x[1:]        -> [20, 30, 40, 50, 60, 70, 80]
```

Example 2

x = [1,2,[10,10.75,[[10,20,30], "mce","pgt"]], 2023, ["hello","palguni"]]

Slices	Slice values
print(len(x))	5
print(len(x[2]))	3
print(len(x[4]))	2
print(x[:])	[1, 2, [10, 10.75, [[10, 20, 30], 'mce', 'pgt']], 2023, ['hello', 'palguni']]
print(x[0])	1
print(x[1]	2
print(x[2])	[10, 10.75, [[10, 20, 30], 'mce', 'pgt']]
print(x[3])	2023
print(x[4])	['hello', 'palguni']
print(x[::1])	[1, 2, [10, 10.75, [[10, 20, 30], 'mce', 'pgt']], 2023, ['hello', 'palguni']]
print(x[::-1])	[['hello', 'palguni'], 2023, [10, 10.75, [[10, 20, 30], 'mce', 'pgt']], 2, 1]
print(x[::2])	[1, [10, 10.75, [[10, 20, 30], 'mce', 'pgt']], ['hello', 'palguni']]
print(x[::-2])	[['hello', 'palguni'], [10, 10.75, [[10, 20, 30], 'mce', 'pgt']], 1]
print(x[2][0])	10
print(x[2][1])	10.75
print(x[2][2])	[[10, 20, 30], 'mce', 'pgt']
print(x[2][1:3])	[10.75, [[10, 20, 30], 'mce', 'pgt']]
print(x[2][::-1])	[[[10, 20, 30], 'mce', 'pgt'], 10.75, 10]
print(x[2][2][:])	[[10, 20, 30], 'mce', 'pgt']
print(x[2][2][::-1])	['pgt', 'mce', [10, 20, 30]]
print(x[2][2][0])	[10, 20, 30]
print(x[2][2][0][::-1])	[30, 20, 10]
print(x[2][2][0][1:3])	[20, 30]
print(x[2][2][1])	mce
print(x[2][2][2])	pgt
print(x[4][0][::-1])	olleh
print(x[4][1][::-2])	iulp

29

List Comprehension

List comprehension is a concise way to create lists in Python by applying an expression to each item in an iterable (like a list, tuple, or range) and then collecting the results in a new list. It provides a compact and readable syntax to generate lists without the need for explicit loops. List comprehensions are powerful and versatile tools in Python, enabling you to create lists with complex expressions and conditions in a succinct manner.

The basic syntax of list comprehension is:

```
new_list = [expression for item in iterable]
```

Here's an example to illustrate list comprehension:

```
# Generating a list of squares using a for loop
squares_using_loop = []
for x in range(1, 6):
   squares_using_loop.append(x ** 2)

# Generating a list of squares using list comprehension
squares_using_comprehension = [x ** 2 for x in range(1, 6)]

print("Squares using loop:", squares_using_loop)
print("Squares using comprehension:", squares_using_comprehension)
```

Output

```
Squares using loop: [1, 4, 9, 16, 25]
Squares using comprehension: [1, 4, 9, 16, 25]
```

You can also include conditions (filtering) in list comprehensions using an if statement. Here's an example:

```
# Generating a list of even squares using list comprehension with a condition
even_squares = [x ** 2 for x in range(1, 6) if x % 2 == 0]
print("Even squares:", even_squares)
```

In this example, only the squares of even numbers from 1 to 5 are included in the even_squares list.

Example: List Comprehension

Sl. No	List Comprehension	Output
1	[x for x in range(5)]	[0, 1, 2, 3, 4]
2	[x*2 for x in range(4)]	[0, 2, 4, 6]
3	[c.upper() for c in "hello"]	['H', 'E', 'L', 'L', 'O']
4	[len(word) for word in ["cat", "dog", "elephant"]]	[3, 3, 8]
5	[(x, y) for x in range(3) for y in range(2)]	[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
6	[row[0] for row in matrix]	Output depends on the matrix used.
7	[num for num in range(10) if num % 2 == 0]	[0, 2, 4, 6, 8]
8	[x + y for x in list1 for y in list2]	Output depends on list1 and list2.
9	[word[::-1] for word in words]	['tac', 'god', 'tnahpele']
10	[num if num % 2 == 0 else "odd" for num in range(6)]	[0, 'odd', 2, 'odd', 4, 'odd']
11	[x for x in range(1, 11) if x % 2 == 0]	[2, 4, 6, 8, 10]
12	[char.upper() for word in sentence.split() for char in word]	Depends on sentence.
13	[x**2 for x in range(5)]	[0, 1, 4, 9, 16]
14	[len(word) for word in sentence.split()]	Output depends on sentence.
15	[num * 2 for num in range(1, 6)]	[2, 4, 6, 8, 10]
16	[char for word in words if len(word) > 3 for char in word]	Depends on words.
17	[x * y for x in range(1, 4) for y in range(1, 4)]	[1, 2, 3, 2, 4, 6, 3, 6, 9]
18	[num for num in range(1, 11) if num % 2 != 0]	[1, 3, 5, 7, 9]
19	[c for c in "hello" if c.isalpha()]	['h', 'e', 'l', 'l', 'o']
20	[x**2 if x % 2 == 0 else x for x in range(1, 6)]	[1, 4, 3, 16, 5]
21	[num for num in range(1, 11) if num % 3 == 0]	[3, 6, 9]
22	[word.upper() for word in sentence.split()]	Output depends on sentence.

23	[len(word) for word in words if len(word) % 2 == 0]	Depends on words.
24	[char * 3 for char in "python"]	['ppp', 'yyy', 'ttt', 'hhh', 'ooo', 'nnn']
25	[x**3 if x % 2 == 0 else x**2 for x in range(1, 6)]	[1, 4, 9, 16, 25]
26	[word[:3] for word in sentence.split()]	Output depends on sentence.
27	[num for num in range(1, 21) if num % 2 == 0 or num % 5 == 0]	[2, 4, 5, 6, 8, 10, 12, 15, 16, 18, 20]
28	[char for char in "programming" if char not in "aeiou"]	['p', 'r', 'g', 'r', 'm', 'm', 'n', 'g']
29	[x*y for x in range(2, 5) for y in range(1, 4)]	[2, 4, 6, 3, 6, 9, 4, 8, 12]
30	[word for word in words if len(word) >= 5 and word.endswith("s")]	Depends on words.
31	[num if num % 2 == 0 else -num for num in range(1, 6)]	[-1, 2, -3, 4, -5]
32	[char.upper() if index % 2 == 0 else char.lower() for index, char in enumerate("listcomp")]	'LiStCoMp'
33	[str(x) + str(y) for x in range(1, 4) for y in range(3, 6)]	['13', '14', '15', '23', '24', '25', '33', '34', '35']
34	[word[::-1] for word in sentence.split() if len(word) % 2 == 0]	Output depends on sentence.
35	[x for x in range(1, 11) if all(x % i != 0 for i in range(2, int(x**0.5) + 1))]	[2, 3, 5, 7]

30

Strings

In Python, a string is a sequence of characters enclosed within single (') or double (" ") quotes. Strings are widely used to represent textual data and can contain letters, numbers, symbols, and whitespace.

Here are some examples that illustrate the concept of strings in Python:

Sl. No	Concept	Examples
1	**Creating Strings**: You can create strings by enclosing characters in either single or double quotes.	single_quoted = **'This is a single-quoted string.'** double_quoted = **"This is a double-quoted string."**
2	**Escaping Characters:** You can use escape characters (prefixed with a backslash) to include special characters within a string.	escaped_string = "**She said, \"Hello!\"**" newline_string = **"Line 1\n Line 2"**
3	**Multiline Strings**: Triple quotes (''' or """) are used to create multiline strings, which can span across multiple lines.	multiline = **'''This is a multiline string.'''**
4	**String Concatenation:** Strings can be concatenated using the + operator.	first_name = "John" last_name = "Doe" full_name = first_name + " " + last_name **# Produces** : "John Doe"
5	**String Replication:** You can replicate a string using the ***** operator.	repeated = "abc" * 3 **# Produces:** "abcabcabc"
6	**String Indexing and Slicing:** Strings are sequences of characters, and you can access individual characters using index positions. Indexing starts from 0.	word = "Python" print(word[0]) # Output: "P" print(word[2:5]) # Output: "tho" print(word[:4]) # Output: "Pyth" print(word[2:]) # Output: "thon" print(word[-1]) # Output: "n"

7	**String Methods:** Strings have built-in methods for various operations, such as transforming case, finding substrings, replacing, and more.	message = "Hello, World!" print(message.upper()) **# Output: "HELLO, WORLD!"** print(message.lower()) **# Output: "hello, world!"** print(message.startswith("H")) **# Output: True** print(message.find("World")) **# Output: 7** print(message.replace("World", "Python")) **# Output: "Hello, Python!"**
8	**String Formatting:** String formatting allows you to insert values into strings in a formatted manner.	name = "Alice" age = 30 formatted_string = **f**"My name is {**name**} and I'm {**age**} years old."

These are just a few examples showcasing the use of strings in Python. Strings are fundamental data types used for various purposes, including text processing, manipulation, and representation.

String Manipulating Functions: String manipulating functions are built-in functions in programming languages that allow you to perform various operations on strings **(sequences of characters)**. These functions help you modify, extract, transform, and manipulate strings according to your requirements. They are essential tools for tasks like searching, replacing, formatting, and cleaning up strings. Here are some examples of string manipulating functions in Python:

len(string): Returns the length of the string.

```
text = "Hello, world!"
length = len(text)  # 13
```

string.lower() and string.upper(): Converts the string to lowercase or uppercase.

```
message = "Python is Amazing"
lower_case = message.lower()  # "python is amazing"
upper_case = message.upper()  # "PYTHON IS AMAZING"
```

string.replace(old, new): Replaces all occurrences of a substring with another substring.

```
sentence = "I like cats, but I prefer dogs."
modified_sentence = sentence.replace("cats", "birds")
# "I like birds, but I prefer dogs."
```

string.split(separator): Splits the string into a list of substrings based on the provided separator.

```
names = "Alice,Bob,Charlie"
name_list = names.split(",")  # ['Alice', 'Bob', 'Charlie']
```

string.join(iterable): Joins a sequence of strings into a single string using the specified string as a separator.

```
words = ["Hello", "world", "Python"]
joined_string = " ".join(words)                    # "Hello world
Python"
```

string.startswith(prefix) and string.endswith(suffix): Checks if the string starts with a given prefix or ends with a given suffix.

```
title = "Python Programming"
starts_with = title.startswith("Python")           # True
ends_with = title.endswith("Programming")          # True
```

string.strip(), string.lstrip(), string.rstrip(): Removes leading/trailing/both whitespace characters from the string.

```
data = "    example    "
stripped = data.strip()                            # "example"
```

****string.format(*args, kwargs):** Formats the string by replacing placeholders with values from arguments or keyword arguments.

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

f-strings (formatted string literals, Python 3.6+): Provides a concise way to embed expressions inside string literals.

```
item = "book"
count = 5
description = f"I have {count} {item}s."                    # "I have 5 books."
```

Following table illustrates the list of some most important python functions, syntax and examples

Sl. No	String Function	Syntax	Example
1	len()	len(string)	length = len("Hello")
2	.upper()	string.upper()	"hello".upper() -> "HELLO"
3	.lower()	string.lower()	"Hello".lower() -> "hello"
4	.capitalize()	string.capitalize()	"hello world".capitalize() -> "Hello world"
5	.title()	string.title()	"hello world".title() -> "Hello World"
6	.swapcase()	string.swapcase()	"Hello World".swapcase() -> "hELLO wORLD"
7	.strip()	string.strip([characters])	" hello ".strip() -> "hello"
8	.rstrip()	string.rstrip([characters])	"hello ".rstrip() -> "hello"
9	.lstrip()	string.lstrip([characters])	" hello".lstrip() -> "hello"
10	.replace()	string.replace(old, new)	"Hello, World!".replace("World", "Python") -> "Hello, Python!"
11	.count()	string.count(substring)	"hello hello".count("hello") -> 2
12	.find()	string.find(substring)	"hello world".find("world") -> 6
13	.index()	string.index(substring)	"hello world".index("world") -> 6
14	.startswith()	string.startswith(prefix)	"hello world".startswith("hello") -> True
15	.endswith()	string.endswith(suffix)	"hello world".endswith("world") -> True

16	.split()	string.split([separator])	"apple,banana,cherry".split(",") -> ['apple', 'banana', 'cherry']
17	.splitlines()	string.splitlines()	"Line 1\nLine 2".splitlines() -> ['Line 1', 'Line 2']
18	.join()	separator.join(iterable)	",".join(['apple', 'banana', 'cherry']) -> "apple,banana,cherry"
19	.isalpha()	string.isalpha()	"hello".isalpha() -> True
20	.isdigit()	string.isdigit()	"123".isdigit() -> True
21	.isalnum()	string.isalnum()	"hello123".isalnum() -> True
22	.isspace()	string.isspace()	" ".isspace() -> True
23	.title()	string.title()	"hello world".title() -> "Hello World"
24	.startswith()	string.startswith(prefix)	"hello world".startswith("hello") -> True
25	.endswith()	string.endswith(suffix)	"hello world".endswith("world") -> True
26	.isupper()	string.isupper()	"HELLO".isupper() -> True
27	.islower()	string.islower()	"hello".islower() -> True
28	.istitle()	string.istitle()	"Hello World".istitle() -> True
29	.find()	string.find(substring)	"hello world".find("world") -> 6
30	.index()	string.index(substring)	"hello world".index("world") -> 6
31	.replace()	string.replace(old, new)	"Hello, World!".replace("World", "Python") -> "Hello, Python!"
32	.count()	string.count(substring)	"hello hello".count("hello") -> 2
33	.center()	string.center(width, [fillchar])	"hello".center(10) -> " hello "
34	.zfill()	string.zfill(width)	"42".zfill(5) -> "00042"
35	.startswith()	string.startswith(prefix)	"hello world".startswith("hello") -> True

36	.endswith()	string.endswith(suffix)	"hello world".endswith("world") -> True
37	.isalpha()	string.isalpha()	"hello".isalpha() -> True
38	.isdigit()	string.isdigit()	"123".isdigit() -> True
39	.isalnum()	string.isalnum()	"hello123".isalnum() -> True
40	.isspace()	string.isspace()	" ".isspace() -> True
41	.join()	separator.join(iterable)	",".join(['apple', 'banana', 'cherry']) -> "apple,banana,cherry"
42	.split()	string.split([separator])	"apple,banana,cherry".split(",") -> ['apple', 'banana', 'cherry']
43	.splitlines()	string.splitlines()	"Line 1\nLine 2".splitlines() -> ['Line 1', 'Line 2']
44	.encode()	string.encode(encoding)	"hello".encode("utf-8")

Consider a string **x = "Contextual Python"**. Here are the values of the given slice expressions for the string x = "Contextual Python":

x[:] (All characters): "Contextual Python"
x[::] (All characters with default step): "Contextual Python"
x[::-1] (All characters in reverse order): "nohtyP lautotsnoC"
x[0:16:1] (All characters from index 0 to 15 with step 1): "Contextual Pytho"
x[0:16:2] (Every second character from index 0 to 15): "Cnoua yhn"
x[0:16:3] (Every third character from index 0 to 15): "Ctayn"
x[0:16:4] (Every fourth character from index 0 to 15): "Cu h"

In slice expressions, the format is x[start:stop:step], where:

- start is the index to start slicing from (inclusive).
- stop is the index to stop slicing at (exclusive).
- step is the step size between elements in the slice. If not provided, it defaults to 1

31

Format Operators

In Python, there are multiple ways to format and display output when using the print statement or function. Three common methods for string formatting are the % operator, the str.format() method, and f-strings (formatted string literals). Each method has its own syntax and use cases. Let's explore them with examples:

1. % Operator (String Interpolation)

The % operator is an older way of formatting strings in Python. It is often referred to as "string interpolation." You use placeholders in the string and then provide values to replace those placeholders using the % operator.

```
name = "Alice"
age = 30

# Using the % operator
print("My name is %s, and I am %d years old." % (name, age))
```

Output
My name is Alice, and I am 30 years old.

In this example

%s is a placeholder for a string.
%d is a placeholder for an integer.
You provide the values to replace these placeholders in the same order in which they appear in the string.

2. str.format() Method

The str.format() method is a more flexible and modern way of formatting strings. It uses placeholders enclosed in curly braces {} and allows you to specify the order of replacement using positional or keyword arguments.

```
name = "Bob"
```

```
age = 25

# Using str.format()
print("My name is {}, and I am {} years old.".format(name, age))
```

Output
My name is Bob, and I am 25 years old.

You can also use positional and keyword arguments to specify the values for placeholders, providing more control over the formatting:

```
# Using positional arguments
print("My name is {0}, and I am {1} years old.".format(name, age))

# Using keyword arguments
print("My name is {name}, and I am {age} years old.".format(name=name, age=age))
```

3. f-strings (Formatted String Literals)

f-strings are a concise and Pythonic way to format strings. They were introduced in Python 3.6 and provide a more readable and convenient way to interpolate variables directly into string literals.

```
name = "Eve"
age = 22

# Using f-strings
print(f"My name is {name}, and I am {age} years old.")
```

Output
My name is Eve, and I am 22 years old.

In an f-string, you place an f or F character before the string literal, and you can directly embed variables and expressions within curly braces {}. Python evaluates the expressions and inserts their values into the string at runtime.

f-strings offer the advantages of readability, clarity, and simplicity for string formatting.

In summary, Python provides several methods for string formatting: % operator, str.format() method, and f-strings (f"..."). You can choose the one that suits your coding style and readability preferences. F-strings are recommended for Python 3.6 and later due to their readability and convenience.

32

Tuples

A tuple in Python is an ordered, immutable collection of elements. Tuples are like lists, but they cannot be modified once created. They are defined using parentheses () and can hold a mix of different data types, including numbers, strings, and even other tuples.

Here's an explanation of tuples with examples and applications:

Creating Tuples: You can create a tuple by enclosing elements in parentheses. Elements are separated by commas.

```
# Creating tuples
empty_tuple = ()
single_item_tuple = (42,)
fruit_tuple = ("apple", "banana", "orange")
mixed_tuple = ("Alice", 25, True)
```

Accessing Elements: You can access elements of a tuple using indexing, just like with lists.

```
fruits = ("apple", "banana", "orange")
print(fruits[0])  # "apple"
print(fruits[2])  # "orange"
```

Tuples are Immutable: Unlike lists, once a tuple is created, you cannot modify its elements. You can't add, remove, or change elements.

```
fruits = ("apple", "banana", "orange")
# fruits[0] = "pear"  # This will result in an error
```

Tuple Unpacking: You can unpack a tuple into variables, which can be useful when returning multiple values from a function.

```
name, age = ("Alice", 30)
print(name)  # "Alice"
print(age)   # 30
```

Applications of Tuples

Returning Multiple Values: Functions can return multiple values as a tuple. This is particularly useful when you want to return different pieces of information together.

```
def get_name_and_age():
   return "Alice", 30

name, age = get_name_and_age()
```

Data Integrity: Tuples can be used to ensure the integrity of data. If you want to represent data that should not be changed, using a tuple can prevent accidental modifications.

Dictionary Keys: Since tuples are immutable, they can be used as keys in dictionaries, unlike lists.

```
coordinates =
{
(10, 20): "Location A",
 (5, 15): "Location B"
}
```

Database Records: Tuples can be used to store database records. Each tuple element can represent a different field in the record.

```
user_record = ("Alice", "alice@example.com", 25)
```

Unordered Operations: In cases where order doesn't matter, and you want to prevent modifications, tuples are a good choice. For example, storing RGB color values.

```
white = (255, 255, 255)
```

Multiple Data Types: Tuples can hold elements of different data types, making them suitable for scenarios where you need to group heterogeneous data.

```
student_info = ("Alice", 25, [95, 87, 92])
```

Tuples are a versatile data structure, and their immutability makes them suitable for situations where you want to store data that shouldn't change after creation or when you want to group related information together.

33

Python References

In Python, variables work as references to objects in memory. Understanding references in Python is essential because it affects how data is stored, shared, and manipulated. Let's explore the concept of references in Python with examples:

Assigning Variables

When you create a variable and assign it a value, you're creating a reference to an object in memory. For example:

```
x = 42
```

Here, x is a reference to the integer object 42.

Multiple References

You can have multiple variables referencing the same object:

```
a = 10
b = a  # Both a and b reference the same integer object 10.
```

Changes to a will affect b, and vice versa because they both refer to the same object.

Lists and References

Lists in Python also work with references. When you create a list and assign it to another variable, both variables reference the same list:

```
list1 = [1, 2, 3]
list2 = list1  # Both list1 and list2 reference the same list object.
```

Changes made to list1 will be reflected in list2, and vice versa.

Function Parameters

In Python, function parameters are references to objects. When you pass an object as an argument to a function, you're passing a reference to that object. Here's an example:

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

original_list = [1, 2, 3]
modify_list(original_list)
```

In this case, modify_list modifies the original list because it received a reference to the same list object.

Objects and Classes

Python's object-oriented nature relies heavily on references. When you create an instance of a class, you're creating a reference to that object:

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

person1 = Person("Alice")
person2 = person1  # Both person1 and person2 reference the same Person object.
```

Here, both person1 and person2 reference the same Person object.

Copying vs. Referencing

Python offers ways to create copies of objects instead of referencing the same object. For example, you can create a shallow copy of a list using slicing:

```
list1 = [1, 2, 3]
list2 = list1[:]  # Create a shallow copy of list1.
```

Now, list1 and list2 are separate objects, and changes to one won't affect the other.

Understanding references is crucial in Python to avoid unexpected behavior, especially when working with mutable objects like lists and dictionaries. It helps you manage how data is shared and modified across different parts of your code.

34

Dictionaries

Dictionaries in Python: A dictionary is a built-in data structure in Python that allows you to store and manage data in key-value pairs. Each key in a dictionary is unique, and it maps to a corresponding value. Dictionaries are unordered collections, meaning the order of the key-value pairs isn't guaranteed to be maintained.

Creating Dictionaries: You can create a dictionary using curly braces {} and specifying key-value pairs separated by colons. Here are examples of creating dictionaries:

```
# Creating an empty dictionary
empty_dict = {}

# Creating a dictionary with multiple elements
student =
{
    "name": "Alice",
    "age": 25,
    "grade": "A"
}
```

Accessing and Modifying Values: You can access values in a dictionary by providing the corresponding key in square brackets. You can also modify values using their keys.

```
# Accessing values
name = student["name"]  # "Alice"
age = student["age"]    # 25

# Modifying values
student["grade"] = "B"
```

Adding and Removing Key-Value Pairs: You can add new key-value pairs to a dictionary by assigning a value to a new key. You can also remove key-value pairs using the del statement.

```
# Adding a new key-value pair
```

```
student["school"] = "ABC School"
# Removing a key-value pair
del student["age"]
```

Dictionary Methods: Dictionaries provide several methods for performing various operations:

```
keys = student.keys()        # Returns keys as a view
values = student.values()   # Returns values as a view
items = student.items()      # Returns key-value pairs as a view
```

Examples and Applications

Data Storage and Retrieval: Dictionaries are often used to store data with meaningful identifiers (keys) for easy retrieval.

```
user =
{
        "username": "john_doe",
         "email": "john@example.com"
}
```

Configurations and Settings: Dictionaries are handy for storing configuration settings and preferences.

```
app_config =
{
        "theme": "dark",
         "language": "en"
}
```

Frequency Counting: Dictionaries are used to count occurrences of items in a dataset.

```
text = "hello world"
char_count = {}
for char in text:
   if char in char_count:
      char_count[char] += 1
   else:
      char_count[char] = 1
```

Lookup and Mapping: Dictionaries are effective for mapping one value to another.

```
product_prices =
{
        "apple": 0.5,
        "banana": 0.3
}
price = product_prices["apple"]  # 0.5
```

Data Aggregation: Dictionaries can be used to aggregate data by grouping related information.

```
sales = [
   {"product": "apple", "quantity": 5},
   {"product": "banana", "quantity": 3}
]
product_quantities = {}
for sale in sales:
   product = sale["product"]
   quantity = sale["quantity"]
   if product in product_quantities:
      product_quantities[product] += quantity
   else:
      product_quantities[product] = quantity
```

JSON-like Data: Dictionaries can represent structured data, similar to JSON.

```
person =
{
         "name": "Alice",
        "age": 30,
        "address":
        {
                "street": "123 Main St",
               "city": "Cityville"
        }
}
```

Dictionaries are essential for storing and manipulating data using meaningful labels (keys). They are widely used for tasks involving data organization, lookups, mappings, and aggregations.

Dictionary Manipulating Functions

Sl. No	Function	Example	Output
1	dict.keys()	my_dict = {"a": 1, "b": 2} keys = my_dict.keys()	keys = ["a", "b"]
2	dict.values()	my_dict = {"a": 1, "b": 2} values = my_dict.values()	values = [1, 2]
3	dict.items()	my_dict = {"a": 1, "b": 2} items = my_dict.items()	items = [("a", 1), ("b", 2)]
4	dict.get(key, default)	my_dict = {"a": 1, "b": 2} value = my_dict.get("c", 0)	value = 0
5	dict.setdefault(key, default)	my_dict = {"a": 1, "b": 2} value = my_dict.setdefault("c", 0)	value = 0
6	dict.pop(key, default)	my_dict = {"a": 1, "b": 2} value = my_dict.pop("a", 0)	value = 1
7	dict.popitem()	my_dict = {"a": 1, "b": 2} key, value = my_dict.popitem()	key = "b", value = 2
8	dict.update(other_dict)	my_dict = {"a": 1} other_dict = {"b": 2} my_dict.update(other_dict)	my_dict = {"a": 1, "b": 2}
9	dict.clear()	my_dict = {"a": 1, "b": 2} my_dict.clear()	my_dict = {}
10	len(dict)	my_dict = {"a": 1, "b": 2} length = len(my_dict)	length = 2
11	key in dict	my_dict = {"a": 1, "b": 2} exists = "a" in my_dict	exists = True
12	dict.copy()	my_dict = {"a": 1, "b": 2} new_dict = my_dict.copy()	new_dict = {"a": 1, "b": 2}
13	dict.fromkeys(keys, value)	keys = ["a", "b"] value = 0 new_dict = dict.fromkeys(keys, value)	new_dict = {"a": 0, "b": 0}
14	dict.items()	my_dict = {"a": 1, "b": 2}	items = [("a", 1),

		items = my_dict.items()	("b", 2)]
15	dict.keys()	my_dict = {"a": 1, "b": 2} keys = my_dict.keys()	keys = ["a", "b"]
16	dict.values()	my_dict = {"a": 1, "b": 2} values = my_dict.values()	values = [1, 2]
17	dict.popitem()	my_dict = {"a": 1, "b": 2} key, value = my_dict.popitem()	key = "b", value = 2
18	dict.values()	my_dict = {"a": 1, "b": 2} values = my_dict.values()	values = [1, 2]
19	dict.popitem()	my_dict = {"a": 1, "b": 2} key, value = my_dict.popitem()	key = "b", value = 2
20	dict.clear()	my_dict = {"a":1,"b":2} my_dict.clear()	my_dict = {}
21	len(dict)	my_dict = {"a": 1, "b": 2} length = len(my_dict)	length = 2
22	key in dict	my_dict = {"a": 1, "b": 2} exists = "a" in my_dict	exists = True
23	dict.copy()	my_dict = {"a": 1, "b": 2} new_dict = my_dict.copy()	new_dict = {"a": 1, "b": 2}
24	dict.fromkeys(keys, value)	keys = ["a", "b"] value = 0 new_dict = dict.fromkeys(keys, value)	new_dict = {"a": 0, "b": 0}
25	dict.clear()	my_dict = {"a": 1, "b": 2} my_dict.clear()	my_dict = {}

35

Sets

Sets in Python: A set is an unordered collection of unique elements in Python. Sets are used to store a collection of distinct items, and they are particularly useful for membership testing and eliminating duplicate values from a sequence. Sets are defined using curly braces {}.

Creating Sets: You can create a set by enclosing elements in curly braces, or by using the **set()** constructor. Here are examples of creating sets:

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

# Creating a set with multiple elements
fruits = {"apple", "banana", "orange"}

# Using the set() constructor
colors = set(["red", "green", "blue"])
```

Adding and Removing Elements: You can add elements to a set using the add() method, and you can remove elements using the remove() method.

```
fruits = {"apple", "banana"}
fruits.add("orange")
fruits.remove("apple")
```

Set Operations: Sets support various set operations such as union, intersection, and difference.

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

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

Set Methods: Python sets provide several methods for performing operations on sets:

```
# Union using the union() method
    union = set1.union(set2)

# Intersection using the intersection() method
    intersection = set1.intersection(set2)

# Difference using the difference() method
    difference = set1.difference(set2)
```

36

Sets Examples and Applications

Removing Duplicates: Sets are useful for removing duplicate values from a sequence, such as a list.

```
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)
```

Membership Testing: Sets are designed for efficient membership testing.

```
student_names = {"Alice", "Bob", "Charlie"}

if  "Alice" in student_names:
    print("Alice is in the set")
```

Mathematical Set Operations: Sets are commonly used to perform mathematical set operations.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union = set1.union(set2)
intersection = set1.intersection(set2)
```

Removing Duplicates from Text: Sets can be used to extract unique words from a text document.

```
text = "This is a sample text. This text has repeated words."
words = set(text.split())
```

Data Filtering: Sets can be used to filter out specific elements from a dataset.

```
all_products = {"apple", "banana", "orange", "grape"}
selected_products = {"apple", "banana"}
filtered_products = all_products - selected_products
```

Membership Check in Social Networks: Sets can be used to determine common friends or connections in social networks.

```
my_connections = {"Alice", "Bob", "Charlie"}
friend_connections = {"Alice", "David", "Eve"}
common_friends = my_connections & friend_connections
```

Set-based Operations in Database Queries: Sets can be used to perform set-based operations in database queries.

```
approved_users = {"Alice", "Charlie", "David"}
users_query = "SELECT * FROM users WHERE username IN
{}".format(tuple(approved_users))
```

Sets are versatile data structures in Python that are particularly useful for managing collections of distinct and unordered items. They have applications in data processing, filtering, and various operations involving unique elements.

37

Set Operations

Set operations in Python involve various operations that you can perform on sets to manipulate their elements and relationships. Here are the common set operations with explanations and examples:

Union (union() or |): The union of two sets A and B is a set containing all the elements from both A and B without duplicates.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result_set = set1.union(set2)

# Alternatively: result_set = set1 | set2
print(result_set)  # Output: {1, 2, 3, 4, 5}
```

Intersection (intersection() or &): The intersection of two sets A and B is a set containing elements that are present in both A and B.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result_set = set1.intersection(set2)
# Alternatively: result_set = set1 & set2
print(result_set)  # Output: {3}
```

Difference (difference() or -): The difference between two sets A and B (A - B) is a set containing elements that are in A but not in B.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result_set = set1.difference(set2)
# Alternatively: result_set = set1 - set2
print(result_set)  # Output: {1, 2}
```

Symmetric Difference (symmetric_difference() or ^): The symmetric difference between two sets A and B is a set containing elements that are in either A or B, but not in both.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result_set = set1.symmetric_difference(set2)
# Alternatively: result_set = set1 ^ set2
print(result_set)  # Output: {1, 2, 4, 5}
```

Subset (issubset()): Checks if one set is a subset of another set. A set A is a subset of B if all elements of A are also in B.

Example

```
set1 = {1, 2}
set2 = {1, 2, 3, 4}
is_subset = set1.issubset(set2)
print(is_subset)  # Output: True
```

Superset (issuperset()): Checks if one set is a superset of another set. A set A is a superset of B if all elements of B are also in A.

Example

```
set1 = {1, 2, 3, 4}
set2 = {1, 2}
is_superset = set1.issuperset(set2)
print(is_superset)  # Output: True
```

These are some of the basic set operations available in Python. They are used to perform common set manipulations and comparisons efficiently.

38

Sets Manipulation Functions

Sl. No	Function	Example	Output
1	add()	my_set = {1, 2, 3} my_set.add(4)	{1, 2, 3, 4}
2	remove()	my_set = {1, 2, 3} my_set.remove(2)	{1, 3}
3	discard()	my_set = {1, 2, 3} my_set.discard(2)	{1, 3}
4	pop()	my_set = {1, 2, 3} removed_item = my_set.pop()	removed_item can be 1, 2, or 3
5	clear()	my_set = {1, 2, 3} my_set.clear()	{}
6	update() or union()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.update(set2)	{1, 2, 3, 4, 5}
7	intersection_update()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.intersection_update(set2)	{3}
8	difference_update()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.difference_update(set2)	{1, 2}
9	symmetric_difference_update()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.symmetric_difference_update(set2)	{1, 2, 4, 5}
10	intersection()	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.intersection(set2)	{3}

11	difference()	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.difference(set2)	{1, 2}
12	symmetric_difference()	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.symmetric_difference(set2)	{1, 2, 4, 5}
13	union()	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.union(set2)	{1, 2, 3, 4, 5}
14	issubset()	set1 = {1, 2} set2 = {1, 2, 3, 4} is_subset = set1.issubset(set2)	True
15	issuperset()	set1 = {1, 2, 3, 4} set2 = {1, 2} is_superset = set1.issuperset(set2)	True
16	copy()	set1 = {1, 2, 3} set2 = set1.copy()	set2 is a copy of {1, 2, 3}
17	clear()	my_set = {1, 2, 3} my_set.clear()	my_set is now an empty set: {}
18	isdisjoint()	set1 = {1, 2, 3} set2 = {4, 5, 6} is_disjoint = set1.isdisjoint(set2)	True (no common elements)
19	remove()	my_set = {1, 2, 3} my_set.remove(2)	{1, 3}
20	discard()	my_set = {1, 2, 3} my_set.discard(2)	{1, 3}
21	pop()	my_set = {1, 2, 3} removed_item = my_set.pop()	removed_item can be 1, 2, or 3
22	union() or `	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.union(set2)	

23	intersection() or &	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.intersection(set2)	{3}
24	difference() or -	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.difference(set2)	{1, 2}
25	symmetric_difference() or ^	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.symmetric_difference(set2)	{1, 2, 4, 5}
26	update()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.update(set2)	set1 is updated to {1, 2, 3, 4, 5}
27	intersection_update()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.intersection_update(set2)	set1 is updated to {3}
28	difference_update()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.difference_update(set2)	set1 is updated to {1, 2}
29	symmetric_difference_update()	set1 = {1, 2, 3} set2 = {3, 4, 5} set1.symmetric_difference_update(set2)	set1 is updated to {1, 2, 4, 5}
16	copy()	set1 = {1, 2, 3} set2 = set1.copy()	set2 is a copy of {1, 2, 3}
17	clear()	my_set = {1, 2, 3} my_set.clear()	my_set is now an empty set: {}
18	isdisjoint()	set1 = {1, 2, 3} set2 = {4, 5, 6} is_disjoint = set1.isdisjoint(set2)	True (no common elements)
19	remove()	my_set = {1, 2, 3} my_set.remove(2)	{1, 3}
20	discard()	my_set = {1, 2, 3} my_set.discard(2)	{1, 3}

21	pop()	my_set = {1, 2, 3} removed_item = my_set.pop()	removed_item can be 1, 2, or 3
22	union() or `	set1 = {1, 2, 3} set2 = {3, 4, 5} result_set = set1.union(set2)	

39

Reading and Writing Files

In computer programming, a "file" refers to a named collection of data or information that is stored on a storage medium, such as a hard drive, SSD, or network storage. Files are used to store and organize data for various purposes, and they come in different types and formats depending on their content and usage. Here's an overview of files and some common types:

Text Files: Text files store data in plain text format, where each character is represented by a specific character encoding (e.g., ASCII or UTF-8). They are commonly used for storing human-readable text data, such as configuration files, source code, and documents.

Binary Files: Binary files store data in a format that is not human-readable. They can contain a wide range of data types, including images, audio, video, executables, and more. Binary files are used to store non-textual data.

CSV (Comma-Separated Values) Files: CSV files are a specific type of text file used for tabular data storage. They consist of rows and columns, with each field separated by a comma or other delimiters like tabs or semicolons. CSV files are commonly used for data interchange between applications.

JSON (JavaScript Object Notation) Files: JSON files are text-based data interchange format often used for configuration files and data storage. They represent data in a hierarchical key-value structure.

XML (eXtensible Markup Language) Files: XML files are used to store structured data in a text-based format using custom tags. They are commonly used for configuration files and data exchange between systems.

How to read and write files in Python

Reading Files in Python: Python provides several ways to read files. The most common method uses the open() function and the read() or readline() methods.

```
# Open a file for reading
with open('example.txt', 'r') as file:
```

```
content = file.read()  # Read the entire file
# Alternatively, you can read line by line using a loop
# for line in file:
#     process(line)
```

In this code, 'example.txt' is the name of the file you want to read. The '**r**' argument indicates that you are opening the file for reading. Using a with statement ensures that the file is properly closed after reading.

Writing Files in Python: To write data to a file, you can open it in write mode ('w') or append mode ('a'). Here's how to write to a file:

```
# Open a file for writing
with open('output.txt', 'w') as file:
    file.write("Hello, World!\n")  # Write a string to the file

# To append data to an existing file
with open('output.txt', 'a') as file:
    file.write("This is an appended line.")
```

Remember that opening a file in **write mode ('w')** will overwrite the file's existing content, so use it carefully. If you want to add content without overwriting, use **append mode ('a').**

Python also provides context managers (**the with statement**) to automatically **close files** after reading or writing, ensuring proper resource management.

These are the basics of reading and writing files in Python. Depending on your specific requirements, you may need to work with different file types and use specialized libraries for parsing and processing data in those files.

Here's a table listing common file handling functions in Python, along with their syntax and examples:

Sl. No	Function	Syntax	Example
1	open()	open(file, mode, buffering)	with open('example.txt', 'r') as file: content = file.read()
2	close()	file.close()	file.close()
3	read()	file.read(size=-1)	content = file.read()
4	readline()	file.readline(size=-1)	line = file.readline()
5	readlines()	file.readlines(hint=-1)	lines = file.readlines()

6	write()	file.write(string)	file.write("Hello, World!")
7	writelines()	file.writelines(lines)	file.writelines(['Line 1', 'Line 2'])
8	tell()	file.tell()	position = file.tell()
9	seek()	file.seek(offset, whence=0)	file.seek(0, 0)
10	flush()	file.flush()	file.flush()
11	truncate()	file.truncate(size=None)	file.truncate(100)
12	exists()	os.path.exists(path)	import os exists = os.path.exists('myfile.txt')
13	isfile()	os.path.isfile(path)	python import os isfile = os.path.isfile('myfile.txt')
14	isdir()	os.path.isdir(path)	python import os isdir = os.path.isdir('/myfolder')
15	mkdir()	os.mkdir(path, mode=0o777, *, dir_fd=None)	python import os os.mkdir('newfolder')
16	rmdir()	os.rmdir(path, *, dir_fd=None)	python import os os.rmdir('oldfolder')
17	rename()	os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None)	python import os os.rename('old.txt', 'new.txt')
18	remove()	os.remove(path, *, dir_fd=None)	python import os os.remove('file.txt')

Note: Functions like os.path.exists(), os.path.isfile(), os.path.isdir(), etc., are part of the os module and are used to check file and directory attributes. These are not file operations themselves but are commonly used alongside file handling.

40

Organizing Files

Organizing files in Python involves performing various file management tasks like creating directories, moving files, renaming files, and deleting files. The **os** module and the **shutil** module are commonly used for these purposes. Below are examples of how to organize files in Python:

Creating Directories: The os.mkdir() function can be used to create a new directory.

```
import os
# Create a new directory
os.mkdir('my_folder')
```

Moving Files: The shutil.move() function can be used to move files from one location to another.

```
import shutil
# Move a file to a new directory
shutil.move('file.txt', 'my_folder/')
```

Renaming Files: To rename a file, use the os.rename() function.

```
import os
# Rename a file
os.rename('old_name.txt', 'new_name.txt')
```

Deleting Files: You can use the os.remove() function to delete a file.

```
import os
# Delete a file
os.remove('file_to_delete.txt')
```

Listing Files in a Directory: To list all files in a directory, the **os.listdir()** function can be used as illustrated below:

```
import os
# List all files in the current directory
files = os.listdir('.')
for file in files:
   print(file)
```

Filtering Files by Extension: You can filter files by their extensions using list comprehensions.

```
import os
# List all .txt files in the current directory
txt_files = [file for file in os.listdir('.') if file.endswith('.txt')]
for file in txt_files:
   print(file)
```

Recursively Walking Through Directories: The os.walk() function enables recursive directory traversal.

```
import os
# Recursively list all files and directories
for root, dirs, files in os.walk('.'):
   for file in files:
      print(os.path.join(root, file))
```

Copying Files: The shutil.copy() function can be used to copy files.

```
import shutil
# Copy a file to a new location
shutil.copy('source_file.txt', 'destination_folder/')
```

Deleting Directories: To delete a directory, use the os.rmdir() function (for empty directories) or shutil.rmtree() (to remove directories and their contents).

```
import os
import shutil

# Remove an empty directory
os.rmdir('empty_folder')

# Remove a directory and its contents recursively
shutil.rmtree('directory_to_remove')
```

These are some common file organization tasks in Python. The os and shutil modules provide a wide range of functions for managing files and directories, to automate various file-related operations in Python programs.

41

Debugging

Debugging is the process of identifying and fixing errors or bugs in code. Python provides several tools and techniques to help developers debug their programs effectively. This section discusses the concept of debugging with examples using Python's built-in debugging tools.

1. **Print Statements:** One of the simplest debugging techniques is to use print statements to display variable values or program flow information. This can help user to understand how program is executing.

   Example

```
def divide(a, b):
    result = a / b
    print(f"Dividing {a} by {b} gives {result}")
    return result

result = divide(10, 2)
```

2. **Using pdb (Python Debugger):** Python comes with a built-in interactive debugger called pdb. Breakpoints can be inserted into the code and interactively inspect variables and step through the code.

   Example

```
import pdb

def divide(a, b):
    pdb.set_trace()                # Start debugging session
    result = a / b
    return result

result = divide(10, 2)
```

   When user run this code, it will pause execution at the pdb.set_trace() line, allowing user to interactively inspect variables and step through the code using commands like c (continue), n (next line), s (step into function), and more.

3. **Using IDEs with Debugging Support**: Integrated Development Environments (IDEs) like PyCharm, Visual Studio Code, and others provide advanced debugging features. One can set breakpoints, inspect variables, and use visual debugging tools.

   Example using Visual Studio Code: Set a breakpoint by clicking to the left of the line number. Run Python script in debug mode. The program will stop at the breakpoint, and user can inspect variables in the Debug Sidebar.

4. **Assertions:** Assertions are used to check if a condition holds true at a particular point in the code. If the condition is False, an AssertionError is raised, helping programmer to identify issues.

   Example

```
def divide(a, b):
    assert b != 0, "Division by zero is not allowed"
    result = a / b
    return result

result = divide(10, 0)  # This will raise an AssertionError
```

5. **Logging:** Python's logging module allows you to add detailed log messages to your code. This can help you track the flow of your program and capture relevant information.

   Example

```
import logging
logging.basicConfig(level=logging.DEBUG)

def divide(a, b):
    logging.debug(f"Dividing {a} by {b}")
    result = a / b
    return result

result = divide(10, 2)
```

   The log messages will provide insights into the program's execution.

6. **Using try and except:** You can use try-except blocks to catch and handle exceptions gracefully. This prevents your program from crashing and allows you to handle errors more effectively.

Example

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError as e:
        print(f"Error: {e}")
        result = None
    return result

result = divide(10, 0)
```

These are some of the common techniques for debugging in Python. Effective debugging is an essential skill for any developer, as it helps identify and resolve issues in the code, leading to more robust and reliable software.

42

Object Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of "objects." In OOP, software is designed and structured using objects, which are instances of classes. Objects can contain both data (attributes) and the methods (functions) that operate on that data. Python is an object-oriented programming language, and it fully supports OOP principles. The key OOP concepts in Python with examples are as follows:

1. **Classes and Objects:** In Python, a class is a blueprint or template for creating objects, and an object is an instance of a class. Classes provide a way to define the structure and behavior of objects, allowing programmers to create and manipulate objects based on that blueprint. Let's delve deeper into classes and objects in Python:

   Defining a Class: To define a class in Python, use the class keyword, followed by the class name. Inside a class, define attributes (**data**) and methods (**functions**) that operate on those attributes. Here's a basic example of a class definition:

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

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."
```

In this example

- **Person is the class name**.
- **The __init__ method** is a special method called a constructor. It initializes object attributes when an object is created.
- **self is a reference** to the object being created. It is the first parameter of all instance methods in Python and is used to access and manipulate object attributes.

Creating Objects (Instances): Once a class has been defined, objects (instances) of that class can be created. Objects are instances of the class and contain the attributes and methods defined in the class. The creation of objects is done as follows.

```
# Create two instances of the Person class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
```

In this example, person1 and person2 are objects created from the Person class.

Accessing Attributes and Methods: The attributes and methods of an object can be accessed using the dot notation. For example:

```
print(person1.name)    # Access the 'name' attribute of person1
print(person2.greet())  # Call the 'greet' method of person2.
```

2. **Encapsulation:** Encapsulation is the practice of bundling data (attributes) and methods (functions) that operate on that data into a single unit (i.e., a class). It helps in hiding the internal implementation details and provides an interface to interact with the object.

   Here's an example of encapsulation in Python:

```
class Student:
    def __init__(self, name, roll_number):
        self.__name = name  # Private attribute
        self._roll_number = roll_number  # Protected attribute

    # Public method to get the student's name
    def get_name(self):
        return self.__name

    # Public method to set the student's name
    def set_name(self, name):
        if len(name) > 0:
            self.__name = name

    # Public method to display student details
    def display_details(self):
        print(f"Name: {self.__name}")
```

```
        print(f"Roll Number: {self._roll_number}")
# Creating a Student object
student = Student("Alice", "A12345")

# Accessing public methods to get and set private attributes
student.set_name("Bob")
print("Student's Name:", student.get_name())

# Accessing protected attribute directly (not recommended)
print("Roll Number:", student._roll_number)

# Accessing private attribute directly (not recommended, but possible)
# Note: It's a convention to prefix private attributes with double
underscores, but it's still accessible.
print("Name (Direct Access):", student._Student__name)

# Displaying student details using a public method
student.display_details()
```

In this example

__name is a private attribute, indicated by the double underscores prefix. It's intended to be hidden from external access.

_roll_number is a protected attribute, indicated by a single underscore prefix. While not private, it's considered a convention that it should not be accessed directly from outside the class.

get_name() and set_name() are public methods that provide controlled access to the private __name attribute.

display_details() is a public method that displays the student's details.

Encapsulation ensures that the internal state of an object is not easily modified from outside the class and that interactions with the object are done through well-defined methods.

3. **Inheritance:** Inheritance allows a new class (subclass or derived class) to inherit properties and methods from an existing class (superclass or base class). This promotes code reuse and allows programmers to create specialized classes based on existing ones.

```
class Animal:
```

```
    def  speak(self):
        pass

class  Dog(Animal):
    def  speak(self):
        return "Woof!"

class  Cat(Animal):
    def  speak(self):
       return "Meow!"

dog = Dog()
cat = Cat()

print(dog.speak())          # Output: Woof!
print(cat.speak())          # Output: Meow!
```

4. **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. This concept is often implemented through **method overriding,** where subclass methods provide their own implementations of methods defined in the superclass.

   Method overriding is a concept in object-oriented programming where a subclass provides a specific implementation for a method that is already defined in its superclass. The overridden method in the subclass should have the same name, parameters, and return type as the method in the superclass. This allows you to provide specialized behavior in the subclass while maintaining a common interface.

   Here's an example of method overriding in Python:

```
class Animal:
    def speak(self):
        return "This is a generic animal sound."

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```
# Creating instances of the derived classes
dog = Dog()
cat = Cat()

# Calling the overridden method
print(dog.speak())          # Output: Woof!
print(cat.speak())          # Output: Meow!
```

In this example

"A base class, Animal, contains a method called speak(), which supplies a generic animal sound.

Two subclasses, Dog and Cat, are derived from the Animal class. In both of these subclasses, the speak() method is overridden with their respective implementations.

When instances of Dog and Cat are created, and the speak() method is invoked on them, the overridden method in each subclass is executed, producing the specific sound associated with that animal.

Method overriding enables the implementation of polymorphism, where objects from different classes can be treated as instances of the same superclass. This is a fundamental concept in object-oriented programming and is employed to create flexible and extensible code."

5. **Abstraction**: Abstraction involves simplifying complex reality by modeling classes based on real-world entities. It focuses on the essential attributes and behaviors of objects while hiding the unnecessary details.

   Encapsulation, Abstraction, and Access Modifiers: Python supports encapsulation and abstraction through the use of access modifiers like private (__) and protected (_). These modifiers allow the control of the visibility of attributes and methods within a class. For example:

```
class MyClass:
    def __init__(self):
        self.__private_var = 42
        self._protected_var = 10

    def my_method(self):
        return self.__private_var + self._protected_var
```

In this example, __private_var is a private attribute, and _protected_var is a protected attribute. These are conventionally considered non-public, and their access should be limited.

6. **Method Overloading and Operator Overloading:** Method Overloading and Operator Overloading: Python does not support method overloading in the same way it is supported in some other languages, where multiple methods with the same name but different parameter lists can be defined. However, similar behavior can be achieved in Python using default arguments or variable-length argument lists.

   Method Overloading: Method overloading in Python is not supported in the same way as in some other languages, where multiple methods with the same name but different parameter lists can be defined. However, Python provides a flexible way to achieve similar behavior using default arguments or variable-length argument lists. An example of 'overloading' a method in Python using these techniques is shown below:

   Method Overloading with Default Arguments: In this approach, a single method is defined with default argument values. Depending on the number and types of arguments passed when the method is called, it behaves differently.

```
class Calculator:
   def  add(self, a, b=0, c=0):
       return a + b + c

# Creating an instance of the Calculator class
calculator = Calculator()

# Calling the add method with different numbers of arguments
result1 = calculator.add(5)
result2 = calculator.add(5, 3)
result3 = calculator.add(5, 3, 2)

print(result1)                  # Output: 5
print(result2)                  # Output: 8
print(result3)                  # Output: 10
```

   In this example, the add() method accepts three arguments but provides default values of 0 for b and c. Depending on how many arguments are passed when the method is called, it behaves accordingly.

Method Overloading with Variable-Length Argument Lists

Python allows the use of variable-length argument lists using *args and **kwargs. This enables the definition of a single method that can accept a variable number of arguments.

```
class Calculator:
    def add(self, *args):
        result = 0
        for num in args:
            result += num
        return result

# Creating an instance of the Calculator class
calculator = Calculator()

# Calling the add method with different numbers of arguments
result1 = calculator.add(5)
result2 = calculator.add(5, 3)
result3 = calculator.add(5, 3, 2, 1, 4)

print(result1)  # Output: 5
print(result2)  # Output: 8
print(result3)  # Output: 15
```

In this example, the add() method accepts a variable number of arguments using *args. It iterates through the arguments and calculates their sum.

While Python doesn't have method overloading based on argument types like some statically-typed languages, these techniques allow you to achieve similar functionality by providing different argument options or using variable-length argument lists to handle various cases.

7. **Operator Overloading:** Python allows the definition of special methods (e.g., __add__, __sub__) to specify how operators should behave for instances of custom classes. Here's a simple example of operator overloading:

```
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
```

```
        self.imag = imag

    def __add__(self, other):
        # Overloading the '+' operator for complex number addition
        return ComplexNumber(self.real + other.real, self.imag +
other.imag)

    def __str__(self):
        return f"{self.real} + {self.imag}i"

# Usage
c1 = ComplexNumber(1, 2)
c2 = ComplexNumber(3, 4)
result = c1 + c2                    # Calls the __add__ method
print(result)                       # Output: 4 + 6i
```

8. **Method Overriding:** Method overriding is a feature in object-oriented programming that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. When a method is overridden in a subclass, the subclass version of the method takes precedence over the superclass version when called on instances of the subclass. This customization of the behavior of inherited methods is done to suit the needs of the subclass.

Here's an explanation of method overriding in Python with examples:

1. **Base Class and Subclass**: Let's start by defining a base class (superclass) and a subclass. The base class will have a method that we'll override in the subclass:

```
class Animal:
    def speak(self):
        return "Animal speaks something."

class Dog(Animal):
    pass
```

   In this example, Animal is the base class, and Dog is the subclass that inherits from Animal.

2. **Method Overriding**: To override a method from the superclass in the subclass, define a method with the same name in the subclass. The method

signature (name and parameters) must match the one in the superclass. Here, we override the speak method in the Dog subclass:

```
class Dog(Animal):
    def speak(self):
        return "Dog barks"
```

Now, the **speak** method in the Dog class overrides the speak method in the Animal class.

3. **Using the Override Method:** You can create instances of the subclass and call the overridden method:

```
animal = Animal()
dog = Dog()

print(animal.speak())  # Output: "Animal speaks something"
print(dog.speak())     # Output: "Dog barks"
```

When you call speak on the dog object, it uses the overridden version in the Dog class. When you call speak on the animal object, it uses the version in the Animal class.

4. **Using super() for Method Overriding**: In some cases, it may be desirable to extend the behavior of the superclass method in the subclass while still utilizing the superclass's implementation. This can be achieved using super():

```
class Cat(Animal):
    def speak(self):
        base_sound = super().speak()
        return f"Cat meows and then {base_sound}"

cat = Cat()
print(cat.speak())  # Output: "Cat meows and then Animal speaks
something"
```

In this example, the Cat class extends the speak method by calling super().speak() to get the result from the Animal class and then adding additional behavior.

Method overriding is a powerful mechanism in Python's object-oriented programming that allows you to customize and specialize the behavior of

methods in subclasses while maintaining a common interface with the superclass.

These are the fundamental concepts of Object-Oriented Programming in Python. OOP provides a powerful and organized way to structure code, promote code reusability, and model real-world entities in programs.

43

Interface

In Python, there is no direct concept of interfaces as found in some other programming languages, such as Java or C#. Nevertheless, Python offers a method to achieve similar functionality through abstract base classes (ABCs) and multiple inheritance. Abstract base classes in Python serve as a means to define a shared interface that derived classes should follow, even though Python does not strictly enforce interface implementation.

Here's an explanation of how to create and use interfaces using abstract base classes in Python with an example:

Step 1: Import the abc module: Import the abc module, which provides the ABC (Abstract Base Class) class and the abstract method decorator.

from abc import ABC, abstractmethod

Step 2: Define an Interface (Abstract Base Class): Create an interface by defining a class that is inherited from ABC and using the **@abstractmethod** decorator to specify abstract methods (methods without implementation). These abstract methods represent the interface's contract that implementing classes must follow.

```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
```

In this example, the Shape class defines an interface with two abstract methods, area, and perimeter. Any class that implements this interface must provide concrete implementations for these methods.

Step 3: Implement the Interface in a Class: To implement the interface, create a class that inherits from the interface and provides concrete implementations for all its abstract methods.

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)
```

Here, the Rectangle class implements the **Shape** interface by providing concrete implementations for the area and perimeter methods.

Step 4: Using the Implemented Interface: Create instances of the class that implements the interface and use them as objects that adhere to the interface's contract.

```
rectangle = Rectangle(4, 5)
print("Rectangle Area:", rectangle.area())       # Output: Rectangle Area: 20
print("Rectangle Perimeter:", rectangle.perimeter())
# Output: Rectangle Perimeter: 18
```

Note: Python does not enforce the implementation of abstract methods like some languages do. However, it does provide a mechanism for documenting and indicating the expected interface for derived classes. Attempting to create an instance of a class that doesn't implement all the abstract methods of its base class will result in a **TypeError** at runtime.

44

Docstring

In Python, a docstring is a special type of string literal that appears as the first statement in a module, class, method, or function. Its purpose is to provide documentation, explanations, and descriptions of the code's purpose, behavior, and usage. Docstrings are used to help developers understand and use the code, and they are accessible through various tools like documentation generators and interactive help.

Docstrings are enclosed in triple-quotes (either single or double) and can span multiple lines. There are various conventions for docstrings in Python, but the most common one is the "Google-style" or "PEP 257" style.

Here's how to use docstrings in Python with examples:

1. **Module-level Docstring:** A module-level docstring is used to provide an overview of the entire module's purpose and contents. It appears at the beginning of a Python module file.

```
"""This module contains functions for performing mathematical operations."""

    def  add(a, b):
       """Add two numbers and return the result."""
       return a + b

    def  subtract(a, b):
       """Subtract b from a and return the result."""
       return a - b
```

   The module-level docstring can be accessed using the __doc__ attribute:

```
    import mymodule
    print(mymodule.__doc__)
```

2. **Class Docstring:** A class docstring is used to provide information about the class and its purpose. It appears immediately below the class definition.

```
class Person:
    """Represents a person with a name and age."""

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

3. **Method Docstring**: A method docstring describes the purpose and usage of a method. It appears immediately below the method definition.

```
class Calculator:
    """A simple calculator class."""

    def add(self, a, b):
        """Add two numbers and return the result."""
        return a + b

    def subtract(self, a, b):
        """Subtract b from a and return the result."""
        return a - b
```

You can access method docstrings using the help() function or by accessing the .__doc__ attribute of the method:

```
calc = Calculator()
help(calc.add)
print(calc.add.__doc__)
```

4. **Function Docstring**: A function docstring provides information about the purpose and usage of a function. It appears immediately below the function definition.

```
def greet(name):
    """Greet the user by name."""
    return f"Hello, {name}!"
```

You can access function docstrings in the same way as method docstrings using help() or the .__doc__ attribute.

5. **Multi-line Docstrings**: Docstrings can span multiple lines for more detailed documentation. Here's an example with a multi-line function docstring:

```
def complex_function(a, b):
    """
    This function performs a complex operation.

    Args:
        a (int): The first operand.
        b (int): The second operand.

    Returns:
        int: The result of the complex operation.
    """
    # Function implementation goes here
```

Using descriptive docstrings is a good practice in Python because it helps to understand the code, and tools like **Sphinx** can generate documentation from docstrings for larger projects.

45

__init__( )

In Python, **__init__()** is a special method (also known as a constructor) that is automatically called when an object of a class is created. It is used to initialize the attributes (variables) of an object. The **__init__()** method is one of the most commonly used methods in Python classes and is crucial for setting up the initial state of objects.

Here's an explanation of **__init__()** in Python with examples:

Basic Usage

```
class MyClass:
   def  __init__(self, arg1, arg2):
       self.attr1 = arg1
       self.attr2 = arg2
```

In this example

__init__(self, arg1, arg2) is the constructor method.

- self is a reference to the instance of the class (the object being created).
- arg1 and arg2 are parameters passed to the constructor.
- self.attr1 and self.attr2 are instance attributes that are initialized with the values of arg1 and arg2, respectively.

Creating Objects: To create an object of the class and initialize its attributes using the __init__() method:

```
obj = MyClass(42, "hello")
```

Now, obj is an instance of MyClass with attr1 set to 42 and attr2 set to "hello".

Default Arguments: You can provide default values for constructor arguments to make them optional:

```
class Person:
   def __init__(self, name="Unknown", age=0):
```

```
        self.name = name
        self.age = age
```

In this example, if you create a Person object without providing arguments, it will use the default values:

```
p1 = Person()        # Uses default values
p2 = Person("Alice", 30)
```

Instance Variables vs. Class Variables: Instance variables (like self.attr1 and self.attr2 in the first example) are unique to each instance of a class. Class variables, on the other hand, are shared among all instances of a class. To create a class variable, define it outside the **__init__()** method and use it without self.

```
class MyClass:
    class_var = 10            # Class variable shared among all instances

    def __init__(self, instance_var):
        self.instance_var = instance_var
```

Common Patterns

- Initializing instance variables to default values.
- Performing setup operations that should occur when an object is created.
- Validating and initializing attributes based on input data.

```
class Circle:
    def __init__(self, radius):
        if radius < 0:
            raise ValueError("Radius cannot be negative")
        self.radius = radius
        self.area = 3.14159 * radius**2

circle = Circle(5)
```

The __init__() method is a fundamental part of object-oriented programming in Python, allowing you to create objects with specific initial states and behaviors. It helps ensure that objects of a class are properly configured when they are first created.

46

__str__( )

In Python, the **__str__()** method is a special method (also known as a magic method or dunder method) that is used to define a custom string representation for an object. When you call the **str()** function or use the **print()** function with an object, Python will internally call the object's **__str__()** method to obtain a string representation of that object.

Here's an explanation of **__str__()** with an example in Python:

Basic Usage

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

    def __str__(self):
        return f"MyClass object with value: {self.value}"
```

In this example

- **__str__(self)** is the __str__() method.
- **self** is a reference to the instance of the class.
- Inside the **__str__()** method, you can return a string that represents the object in a human-readable way.

Using __str__() Method: Now, let's create an object of the class and see how the __str__() method is used:

```
obj = MyClass(42)
print(obj)              # Output: MyClass object with value: 42
```

When you call print(obj) or str(obj), Python internally calls **obj.__str__()** to obtain the string representation of the obj object.

Custom String Representation: You can customize the string representation in any way you like, including formatting attributes and including additional information:

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

   def __str__(self):
      return f"Person: {self.name}, Age: {self.age}"
```

Now, when you create a Person object and print it:

```
person = Person("Alice", 30)
print(person)                    # Output: Person: Alice, Age: 30
```

Using __str__() for Debugging: The __str__() method is also helpful for debugging, as it provides a human-readable representation of an object's state. It's a good practice to define __str__() for your classes to aid in debugging and make your code more understandable.

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y

   def __str__(self):
       return f"Point(x={self.x}, y={self.y})"
```

With this implementation, when you print a Point object, you get a clear representation of its coordinates:

```
point = Point(3, 5)
print(point)  # Output: Point(x=3, y=5)
```

47

Walrus Operator in Python

The "**walrus operator**" is a colloquial term for the assignment expression operator (**:=**) introduced in Python 3.8. It allows you to assign a value to a variable as part of an expression. This operator is named "walrus" because its symbol (:=) resembles the eyes and tusks of a walrus.

The primary use case for the walrus operator is to simplify code by both ***assigning a value to a variable and using that value in an expression***. It is especially helpful in situations where you need to avoid redundant calculations or evaluate an expression with side effects, such as in loops and conditionals.

Here's an example to illustrate the use of the walrus operator:

Without Walrus Operator:

```
# Calculate the sum of squares of numbers until the sum exceeds 100
total = 0
num = 1

while total <= 100:
   total += num ** 2
   num += 1

print(total)  # Output: 140
```

In this example, we calculate the sum of squares of numbers until the sum exceeds 100. We need to update the num variable both within the loop condition and inside the loop body.

With Walrus Operator

```
# Calculate the sum of squares of numbers until the sum exceeds 100
total = 0
num = 1

while (total := total + num ** 2) <= 100:
   num += 1
print(total)  # Output: 140
```

In this version, we use the walrus operator (:=) to both update the total variable and check if it exceeds 100 in a single line. This results in more concise and readable code.

The walrus operator is not limited to loops; it can be used in various contexts, including list comprehensions, conditional expressions, and function calls. Here's an example with a list comprehension:

```
# Filter and double values in a list using the walrus operator
    numbers = [1, 2, 3, 4, 5, 6]
    filtered_numbers = [y for x in numbers if (y := x * 2) <= 8]
    print(filtered_numbers)                    # Output: [2, 4, 6, 8]
```

In this example, the walrus operator is used to avoid recomputing the value x * 2 multiple times within the list comprehension.

The walrus operator enhances code readability and can lead to more efficient code by avoiding redundant computations. However, it should be used judiciously to maintain code clarity and avoid excessive assignment expressions within complex expressions.

48

Match Case Statement

In Python 3.10 and later versions, the match statement has been introduced as a powerful and flexible way to perform pattern matching and make decisions based on the structure of data. It allows you to compare an expression to a set of patterns and execute code based on the first matching pattern. The match statement is often used in situations where you would have used a series of if and elif statements in the past, simplifying code and making it more readable.

Here's the basic syntax of the match statement:

```
match expression:
    case pattern1:
        # Code to execute if the expression matches pattern1
    case pattern2:
        # Code to execute if the expression matches pattern2
    case pattern3:
        # Code to execute if the expression matches pattern3
    ...
    case _:
        # Code to execute if no patterns match (optional)
```

Each case block specifies a pattern, and the code within that block is executed if the expression matches that pattern. You can use various patterns, including literals, variables, and data structures. Here's an example of how to use the match statement in Python:

```
def get_day_name(day_number):
    match day_number:
        case 1:
            return "Monday"
        case 2:
            return "Tuesday"
        case 3:
            return "Wednesday"
        case 4:
```

```
            return "Thursday"
        case 5:
            return "Friday"
        case 6:
            return "Saturday"
        case 7:
            return "Sunday"
        case _:
            return "Invalid day number"

day_number = 3
day_name = get_day_name(day_number)
print(day_name)  # Output: "Wednesday"
```

In this example

We define a function get_day_name that takes a day_number as an argument. Inside the function, we use the match statement to compare day_number with different cases.

If day_number matches one of the specified cases (e.g., case 3:), the corresponding code block is executed, and the corresponding day name is returned.

If day_number doesn't match any of the specified cases, the case _: block is executed, which returns "Invalid day number."

In this way, the match statement simplifies the code by providing a concise and readable way to perform pattern matching and make decisions based on the value of an expression.

49

Regular Expressions

In Python, a regular expression, often referred to as a regex or regexp, is a powerful tool for pattern matching and manipulation of strings. It allows you to define a pattern that can be used to search for, match, or manipulate text data. Python's **re** module provides support for regular expressions.

Here's a brief overview of how regular expressions work in Python, followed by an example:

Import the re module: First, you need to import the **re** module to use regular expressions in Python.

```
import re
```

Define a regular expression pattern: You specify a pattern using a combination of regular characters and special metacharacters that define the pattern you want to match.

Use functions from the re module: You can use functions like re.match(), re.search(), re.findall(), re.finditer(), re.sub(), and re.split() to perform various operations with regular expressions. Here's a simple example:

```
import re

# Sample text
text = "Hello, my email address is john.doe@example.com, and my
phone number is 555-123-4567."

# Define a regular expression pattern for finding email addresses
email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b'

# Use re.findall() to find all email addresses in the text
email_addresses = re.findall(email_pattern, text)

# Print the found email addresses
```

```
for email in email_addresses:
    print("Found email:", email)
```

In this example, the regular expression pattern r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b' is used to find email addresses in the text. It looks for patterns that match typical email address formats. The re.findall() function is then used to find all email addresses in the given text.

Output
Found email: john.doe@example.com

This is just a basic example. Regular expressions can be as simple or as complex as your pattern matching needs require. They are a powerful tool for tasks such as data validation, text parsing, and data extraction from strings.

Regular Expressions in Python: Cheat Sheet

Regular expressions, commonly known as **regex**, play a pivotal role in Python programming, as well as in numerous other programming languages. They serve the essential purpose of locating and potentially altering specific text patterns within strings. In the realm of regular expressions, a cluster of characters collaborates to define the sought-after pattern, often referred to as the "reg-ex" pattern.

The challenge with **regex** lies not so much in comprehending its principles, but in recalling the precise syntax and the intricacies of crafting patterns tailored to our specific needs.

To simplify this endeavor, we offer a comprehensive Regex Cheat Sheet that encompasses a wide array of character classes, special symbols, modifiers, sets, and more, all of which are indispensable tools within the realm of regular expressions. This resource serves as a valuable aid for those navigating the intricate landscape of pattern matching and manipulation.

Meta-Characters

Metacharacter	Meaning	Example
.	Match any character except a newline	re.search(r'c.t', 'cat') matches 'cat'
^	Match the start of a string	re.match(r'^Hello', 'Hello, World!') matches 'Hello'
$	Match the end of a	re.search(r'World!$', 'Hello,

	string	World!') matches 'World!'
*****	Match 0 or more repetitions	re.search(r'ab*', 'a') matches 'a'
+	Match 1 or more repetitions	re.search(r'ab+', 'abbb') matches 'abb'
?	Match 0 or 1 repetition	re.search(r'colou?r', 'color') matches 'color'
|	Alternation (OR)	
()	Groups characters for operations	`re.search(r'(cat
[]	Defines a character class	re.search(r'[aeiou]', 'apple') matches 'a'
{}	Specifies a specific number of repetitions	re.search(r'a{2,3}', 'aaa') matches 'aaa'

Character Classes

Character/ Metacharacter	Meaning	Example
\d	Matches any digit (equivalent to [0-9])	re.search(r'\d', 'The number is 42') matches '4'
\D	Matches any non-digit character (equivalent to [^0-9])	re.search(r'\D', 'The number is 42') matches 'T'
\w	Matches any word character (equivalent to [a-zA-Z0-9_])	re.search(r'\w', 'Hello, world!') matches 'H'
\W	Matches any non-word character (equivalent to [^a-zA-Z0-9_])	re.search(r'\W', 'Hello, world!') matches ','
\s	Matches any whitespace character (equivalent to [\t\n\r\f\v])	re.search(r'\s', 'Hello\tworld!') matches '\t'
\S	Matches any non-whitespace character (equivalent to [^ \t\n\r\f\v])	re.search(r'\S', 'Hello\tworld!') matches 'H'
[abc]	Matches any one of the characters within the square brackets	re.search(r'[aeiou]', 'apple') matches 'a'
[^abc]	Matches any character that is not in the square brackets	re.search(r'[^0-9]', 'The price is $42') matches 'T'
[a-z]	Matches any lowercase letter from 'a' to 'z'	re.search(r'[a-z]', 'Hello World') matches 'e'

[A-Z]	Matches any uppercase letter from 'A' to 'Z'	re.search(r'[A-Z]', 'Hello World') matches 'H'
[0-9]	Matches any digit from 0 to 9	re.search(r'[0-9]', 'The answer is 42') matches '4'
[a-zA-Z]	Matches any letter (either lowercase or uppercase)	re.search(r'[a-zA-Z]', '123abc') matches 'a'
[0-9A-Fa-f]	Matches a hexadecimal digit (0-9, A-F, or a-f)	re.search(r'[0-9A-Fa-f]', '1A2b3C') matches '1'
[0-9a-zA-Z_]	Matches any word character, including letters, digits, and underscore	re.search(r'[0-9a-zA-Z_]', 'Hello_123') matches 'H'
[^ \t]	Matches any character that is not a space or tab	re.search(r'[^ \t]', 'Hello World') matches 'H'
\b	Matches a word boundary	re.search(r'\bword\b', 'This is a word.') matches 'word'
\B	Matches a non-word boundary	re.search(r'\Bnot\b', 'This is not a test.') matches 'not'
\A	Matches the start of a string	re.search(r'\AHello', 'Hello, World!') matches 'Hello'
\Z	Matches the end of a string	re.search(r'World\Z', 'Hello, World') matches 'World'
\n	Matches a newline character	re.search(r'Line1\nLine2', 'Line1\nLine2') matches '\n'

Quantifiers

Quantifier	Meaning	Example
*	Matches 0 or more repetitions	re.search(r'ab*', 'a') matches 'a'
+	Matches 1 or more repetitions	re.search(r'ab+', 'abbb') matches 'abb'
?	Matches 0 or 1 repetition	re.search(r'colou?r', 'color') matches 'color'
{n}	Matches exactly n times	re.search(r'a{2}', 'aaa') matches 'aa'

{n,}	Matches n or more times	re.search(r'a{2,}', 'aaaa') matches 'aaaa'
{n,m}	Matches between n and m times	re.search(r'a{2,4}', 'aaaa') matches 'aaaa'
*?	Matches 0 or more repetitions (non-greedy)	re.search(r'ab*?', 'abbb') matches 'a'
+?	Matches 1 or more repetitions (non-greedy)	re.search(r'ab+?', 'abbb') matches 'ab'
??	Matches 0 or 1 repetition (non-greedy)	re.search(r'colou??r', 'color') matches 'col'
{n}?	Matches exactly n times (non-greedy)	re.search(r'a{2}?', 'aaa') matches 'aa'
{n,}?	Matches n or more times (non-greedy)	re.search(r'a{2,}?', 'aaaa') matches 'aa'
{n,m}?	Matches between n and m times (non-greedy)	re.search(r'a{2,4}?', 'aaaa') matches 'aa'

Quantifiers specify how many repetitions of a character or group you want to match in a regular expression. Greedy quantifiers (e.g., *, +, ?, {n}, {n,}, {n,m}) match as much text as possible, while non-greedy quantifiers (e.g., *?, +?, ??, {n}?, {n,}?, {n,m}?) match as little as possible while still satisfying the pattern.

These quantifiers allow you to control the flexibility and specificity of your pattern matching in regular expressions. The examples demonstrate how each quantifier behaves when applied to different strings**.**

Sets

Set	Meaning	Example
[aeiou]	Matches any one vowel (a, e, i, o, or u)	re.search(r'[aeiou]', 'apple') matches 'a'
[0-9]	Matches any digit (0 through 9)	re.search(r'[0-9]', 'The answer is 42') matches '4'
[A-Z]	Matches any uppercase letter (A through Z)	re.search(r'[A-Z]', 'Hello World') matches 'H'
[a-zA-Z]	Matches any letter (either lowercase or uppercase)	re.search(r'[a-zA-Z]', '123abc') matches 'a'
[0-9a-z]	Matches any lowercase letter or digit	re.search(r'[0-9a-z]', 'Hello123') matches 'e'

[^abc]	Matches any character that is not 'a', 'b', or 'c'	re.search(r'[^0-9]', 'The price is $42') matches 'T'
[A-EH-J]	Matches any uppercase letter from 'A' to 'E' or 'H' to 'J'	re.search(r'[A-EH-J]', 'Hello World') matches 'H'
[0-9A-Fa-f]	Matches a hexadecimal digit (0-9, A-F, or a-f)	re.search(r'[0-9A-Fa-f]', '1A2b3C') matches '1'
[^ \t]	Matches any character that is not a space or tab	re.search(r'[^ \t]', 'Hello World') matches 'H'
[0-9A-Za-z]	Matches any alphanumeric character	re.search(r'[0-9A-Za-z]', '@123ABC') matches '1'
[a-zA-Z0-9_]	Matches any word character (letters, digits, or underscore)	re.search(r'[a-zA-Z0-9_]', 'Hello_123') matches 'H'
[^A-Za-z0-9]	Matches any character that is not alphanumeric	re.search(r'[^A-Za-z0-9]', 'Hello_123!') matches '!'
[\t\n\r]	Matches any of the specified whitespace characters	re.search(r'[\t\n\r]', 'Hello\nworld!') matches '\n'
[^\t\n\r]	Matches any character that is not a whitespace character	re.search(r'[^\t\n\r]', 'Hello\nworld!') matches 'H'
[1-5a-c]	Matches any character from '1' to '5' or 'a' to 'c'	re.search(r'[1-5a-c]', 'apple2') matches 'a'
[^XYZ]	Matches any character that is not 'X', 'Y', or 'Z'	re.search(r'[^XYZ]', 'Hello World') matches 'H'
[\d\D]	Matches any character (digit or non-digit)	re.search(r'[\d\D]', 'Hello123') matches 'H'
[\w\W]	Matches any character (word or non-word)	re.search(r'[\w\W]', 'Hello_123') matches 'H'
[\s\S]	Matches any character (whitespace or non-whitespace)	re.search(r'[\s\S]', 'Hello world') matches 'H'

Anchors

Anchor	Meaning	Example
^	Matches the start of a string	re.match(r'^Hello', 'Hello, World!') matches 'Hello'
$	Matches the end of a string	re.search(r'World!$', 'Hello, World!') matches 'World!'
\b	Matches a word boundary	re.search(r'\bword\b', 'This is a word.') matches 'word'

\B	Matches a non-word boundary	re.search(r'\Bnot\b', 'This is not a test.') matches 'not'
\A	Matches the start of a string (like ^, but doesn't work in multiline mode)	re.search(r'\AHello', 'Hello, World!') matches 'Hello'
\Z	Matches the end of a string (like $, but doesn't work in multiline mode)	re.search(r'World\Z', 'Hello, World') matches 'World'
\n	Matches a newline character	re.search(r'Line1\nLine2', 'Line1\nLine2') matches '\n'
\t	Matches a tab character	re.search(r'Tab\tSeparated', 'Tab\tSeparated') matches '\t'
\r	Matches a carriage return character	re.search(r'End\rLine', 'End\rLine') matches '\r'
\f	Matches a form feed character	re.search(r'Page\fBreak', 'Page\fBreak') matches '\f'
\v	Matches a vertical tab character	re.search(r'Page\vBreak', 'Page\vBreak') matches '\v'
\A	Matches the start of a string (like ^, but doesn't work in multiline mode)	re.search(r'\AHello', 'Hello, World!') matches 'Hello'

These anchors allow you to specify positions within a string where a match should occur. The examples demonstrate how each anchor can be used to find text patterns at specific locations within strings.

Modifiers: Modifiers, also known as flags, are used in regular expressions in Python to control various matching behaviors. Here's a list of more than 10 common modifiers, their meanings, and examples:

Modifier	Meaning	Example
re.I	Case-insensitive matching	**re.search(r'apple', 'APPLE', re.I)** matches 'APPLE'
re.M	Multiline mode	**re.search(r'^Line', 'Line1\nLine2', re.M)** matches 'Line'
re.S	Dot matches all (including newline)	**re.search(r'.+', 'Line1\nLine2', re.S)** matches 'Line1\nLine2'
re.X	Extended flag (ignore whitespace and comments)	**re.search(r'Hello World', 'Hello World', re.X)** matches 'HelloWorld'

re.U	Unicode matching	**re.search(r'\u00E9', 'Café', re.U)** matches 'é'
re.L	Locale dependent matching	**re.search(r'\w+', 'café', re.L)** matches 'café'
re.A	ASCII-only matching	**re.search(r'\w+', 'café', re.A)** doesn't match 'café'
re.DEBUG	Debugging mode	**re.search(r'apple', 'apple', re.DEBUG)** provides debugging information
re.IGNORECASE	Case-insensitive matching (alternative)	**re.search(r'apple', 'APPLE', re.IGNORECASE)** matches 'APPLE'
re.MULTILINE	Multiline mode (alternative)	**re.search(r'^Line', 'Line1\nLine2', re.MULTILINE)** matches 'Line'
re.DOTALL	Dot matches all (including newline, alternative)	**re.search(r'.+', 'Line1\nLine2', re.DOTALL)** matches 'Line1\nLine2'
re.VERBOSE	Extended flag (ignore whitespace and comments, alternative)	**re.search(r'Hello World', 'Hello World', re.VERBOSE)** matches 'HelloWorld'

These modifiers can be passed as optional flags when using regular expression functions like re.search() or re.compile(). They allow you to customize the behavior of your regular expressions to meet specific matching requirements. The examples demonstrate how each modifier affects the matching behavior of regular expressions.

Here are the regular expression flags with explanations rewritten for clarity:

a: This flag makes the regular expression pattern match ASCII characters only, excluding non-ASCII characters.

i: When this flag is enabled, the regular expression matching becomes case-insensitive. It means that uppercase and lowercase letters are treated as the same.

L: Enabling this flag causes the regular expression to use locale-specific character classes when matching. It considers the current locale settings for character classification.

m: With this flag turned on, the ^ and $ anchors in the regular expression pattern match the start and end of each line within the text, making it useful for multi-line matching.

s: When this flag is set, the dot . in the regular expression pattern matches everything, including newline characters. In essence, it makes the dot "dot-all."

u: Enabling this flag allows the regular expression to match Unicode character classes. It ensures that Unicode characters are treated correctly in character class expressions.

x: With this flag active, the regular expression ignores whitespace and allows for comments. This makes complex patterns more readable by permitting spaces and comments within the pattern.

Examples

Here are more than 25 commonly used regular expressions in Python for various purposes, along with their meanings, presented in rows and columns:

Regular Expression	Meaning	Example
Email Address:		
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$	Matches a valid email address.	re.match(email_pattern, 'user@example.com')
Date of Birth:		
`^(0[1-9]	1[0-2])/(0[1-9]	[12][0-9]
IP Address:		
^(\d{1,3}\.){3}\d{1,3}$	Matches a valid IPv4 address.	re.match(ip_pattern, '192.168.0.1')
Name of Person:		
^[A-Z][a-z]+ [A-Z][a-z]+$	Matches a full name with an initial and capitalization.	re.match(name_pattern, 'John Doe')
Date Format:		
`^(0[1-9]	1[0-2])/(0[1-9]	[12][0-9]
Mobile Number:		
^[0-9]{10}$	Matches a 10-digit mobile phone number.	re.match(phone_pattern, '1234567890')

Landline Number:		
^\d{3}-\d{3}-\d{4}$	Matches a standard US phone number in the format "123-456-7890".	re.match(landline_pattern, '123-456-7890')
URL:		
`^(https?	ftp)://[^\s/$.?#].[^\s]*$ `	Matches a valid URL starting with "http://", "https://", or "ftp://".
ZIP Code:		
^\d{5}(-\d{4})?$	Matches a US ZIP code in the format "12345" or "12345-6789".	re.match(zip_code_pattern, '12345')
Credit Card Number:		
^\d{4}-\d{4}-\d{4}-\d{4}$	Matches a 16-digit credit card number in the format "0000-0000-0000-0000".	re.match(credit_card_pattern, '1234-5678-9012-3456')
Social Security Number:		
^\d{3}-\d{2}-\d{4}$	Matches a US Social Security Number in the format "123-45-6789".	re.match(ssn_pattern, '123-45-6789')
HTML Tags:		
`<(?:"[^"]"['"]	'[^']['"]	[^'">])+>`
Hex Color Code:		
`^#([A-Fa-f0-9]{6}	[A-Fa-f0-9]{3})$`	Matches a valid hexadecimal color code in the format "#RRGGBB" or "#RGB".
Time in 12-Hour Format:		
`^(1[0-2]	0?[1-9]):[0-5][0-9] (AM	PM)$`
Time in 24-Hour Format:		
`^([01][0-9]	2[0-3]):[0-5][0-9]$`	Matches a time in 24-hour format, e.g., "14:15".

- r"^\d{3}$" matches exactly 3 digits.
- r"\b\w+\b" matches whole words.
- r"\d{3}-\d{2}-\d{4}" matches a social security number.
- r"[A-Z][a-z]+" matches capitalized words.
- r"\d+" matches one or more digits.
- r"(cat|dog)" matches "cat" or "dog".

Functions in Python re Module

Here are more than 10 commonly used functions in the Python re module for working with regular expressions, along with their meanings and examples presented in rows and columns:

Function	Meaning	Example
re.match(pattern, string)	Determines if the regular expression pattern matches at the beginning of the string. If it matches, it returns a match object; otherwise, it returns None.	re.match('abc', 'abcdef') returns a match object.
re.search(pattern, string)	Searches the entire string for a match to the regular expression pattern. Returns a match object if a match is found; otherwise, returns None.	re.search('apple', 'I like apples') returns a match object.
re.findall(pattern, string)	Returns all non-overlapping matches of the regular expression pattern in the string as a list of strings.	re.findall('\d+', 'There are 123 apples and 456 oranges.') returns ['123', '456'].
re.finditer(pattern, string)	Returns an iterator yielding match objects for all non-overlapping matches of the regular expression pattern in the string.	matches = re.finditer('ab', 'abababab'); [m.group() for m in matches] returns ['ab', 'ab', 'ab'].
re.split(pattern, string)	Splits the string by occurrences of the regular expression	re.split(r'\s+', 'Hello World') returns ['Hello', 'World'].

	pattern and returns a list of substrings.	
re.sub(pattern, replacement, string)	Searches the string for all matches of the regular expression pattern and replaces them with the replacement string. Returns the modified string.	re.sub(r'\d', 'X', 'There are 123 apples.') returns 'There are XXX apples.'.
re.subn(pattern, replacement, string)	Similar to re.sub(), but returns a tuple containing the modified string and the number of substitutions made.	re.subn(r'\d', 'X', 'There are 123 apples.') returns ('There are XXX apples.', 3).
re.compile(pattern)	Compiles the regular expression pattern into a regex object, which can be reused for matching.	pattern = re.compile(r'\d+'); pattern.match('123') returns a match object.
re.purge()	Clears the regular expression cache, removing any cached regex objects.	re.purge()
re.escape(string)	Escapes any special characters in the string, making it safe to use as a literal part of a regular expression pattern.	re.escape('special*chars') returns 'special*chars'.
re.fullmatch(pattern, string)	Determines if the entire string matches the regular expression pattern. Returns a match object if it's a full match; otherwise, returns None.	re.fullmatch('abc', 'abc') returns a match object.

50

Difference Between if and if Else Statement

If statement

- The if statement is used to conditionally execute a block of code when a specified condition is True.
- If the condition is True, the code inside the if block is executed.
- There is no alternative code executed if the condition is False.
- It's a simple branching structure for conditional execution.

Example

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

if-else Statement

- The if-else statement is used to conditionally execute one block of code when a specified condition is True and another block when it's False.
- If the condition is True, the code inside the if block is executed. Otherwise, the code inside the else block is executed.
- It provides an alternative path of execution when the condition is not met.

Example

```
age = 16
if age >= 18:
   print("You are an adult.")
else:
   print("You are not an adult.")
```

In the first example using the if statement, the message is printed because the condition age >= 18 is True.

In the second example using the if-else statement, since the condition age >= 18 is False, the message in the else block is printed instead.

In summary, the key difference is that the if statement provides a single branch of code execution based on a condition, while the if-else statement provides two branches, allowing you to handle both the True and False outcomes of the condition.

51

Difference Between for and While Loop

for Loop

- The for loop is used when you know in advance how many times you want to iterate or loop through a sequence.
- It iterates over a sequence (e.g., a list, tuple, string, or range) and executes a block of code for each item in the sequence.
- The loop variable takes on the value of each item in the sequence during each iteration.
- It's particularly useful when you have a collection of items to process or a specific number of iterations.

Example

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

In this example, the for loop iterates over the list of fruits and prints each fruit.

while Loop

- The while loop is used when you want to repeat a block of code as long as a specific condition is True.
- It keeps executing the block of code as long as the condition remains True.
- It's useful when you don't know in advance how many iterations are needed, and the loop continues until a condition is no longer met.

Example

```
count = 1
while count <= 5:
    print(f"Count is {count}")
    count += 1
```

In this example, the while loop repeatedly prints the value of count until it reaches 5.

In summary

Use a for loop when you have a known sequence to iterate over a specific number of times. Use a while loop when you want to keep looping until a certain condition becomes False, and you may not know in advance how many iterations are needed.

52

Difference Between List and Strings

Lists and strings are both data types in Python, but they have some fundamental differences:

Data Type

- A string is a sequence of characters enclosed in single (' ') or double (" ") quotes.
- A list is an ordered collection of items enclosed in square brackets [ ].

Mutability

- Strings are immutable, meaning you cannot change the characters in a string once it's created. You can create a new string with the desired modifications.
- Lists are mutable, so you can modify, add, or remove elements within a list after it's created.

Ordered vs. Unordered

- Strings are ordered, meaning the characters have a specific sequence, and you can access them by their index.
- Lists are ordered as well, and each element has an index, allowing you to access, modify, or reorder elements based on their positions.

Character vs. Element

- In a string, the individual elements are characters.
- In a list, the elements can be of different data types, including strings, numbers, or other objects.

Representation

- Strings are represented as a single sequence of characters, e.g., "Hello, World!".

- Lists are represented as a sequence of items enclosed in square brackets, e.g., [1, 2, 3, 4].

Methods and Operations

- Strings have methods specific to strings, such as split(), join(), and replace().
- Lists have methods specific to lists, such as append(), insert(), and remove().

Examples

String

```
my_string = "Hello, World!"
print(my_string[0])  # Accessing a character
# This will raise an error because strings are immutable:
# my_string[0] = 'h'
```

List

```
my_list = [1, 2, 3, 4]
print(my_list[0])  # Accessing an element
my_list[0] = 5  # Modifying an element
print(my_list)  # Output: [5, 2, 3, 4]
```

In summary, while strings and lists are both sequences in Python, they differ in terms of mutability, the types of elements they can contain, and the operations you can perform on them. Strings are suitable for working with textual data, while lists are versatile for storing collections of items of various types.

53

Difference between Sets and List

In Python, sets and lists are both used to store collections of elements, but they have distinct characteristics and use cases. Here's a differentiation between sets and lists with examples:

1. Order

List: Lists are ordered collections, which means the elements are stored in a specific order, and you can access them by their position (index) in the list.

Set: Sets are unordered collections, which means they do not have a specific order for their elements, and you cannot access elements by index.

Example

```
# List
my_list = [3, 1, 2, 3]
print(my_list[0])  # Accessing by index, prints 3

# Set
my_set = {3, 1, 2, 3}
# You cannot access elements by index in a set.
```

2. Duplicate Elements

List: Lists allow duplicate elements. You can have the same value multiple times in a list.

Set: Sets do not allow duplicate elements. If you try to add a duplicate element, it will be ignored.

Example

```
# List with duplicates
my_list = [1, 2, 2, 3, 3, 3]
print(my_list)  # Prints [1, 2, 2, 3, 3, 3]

# Set without duplicates
```

```
my_set = {1, 2, 2, 3, 3, 3}
print(my_set)  # Prints {1, 2, 3}
```

3. Mutable vs. Immutable

List: Lists are mutable, which means you can change their contents (add, remove, modify elements) after creation.

Set: Sets are mutable, but the elements themselves must be immutable (e.g., numbers, strings, tuples). You can add and remove elements from a set, but you cannot change an element that is already in the set.

Example

```
# List is mutable
my_list = [1, 2, 3]
my_list.append(4)  # Modifying the list
print(my_list)  # Prints [1, 2, 3, 4]

# Set is also mutable
my_set = {1, 2, 3}
my_set.add(4)  # Modifying the set
print(my_set)  # Prints {1, 2, 3, 4}

# However, you cannot change an element in a set:
# my_set[0] = 5  # This will result in an error.
```

4. Membership and Operations

List: Lists are typically used when the order and duplicates matter, and you need to access elements by their index. Lists support various operations like slicing, concatenation, and more.

Set: Sets are used when you need to store a collection of distinct elements, and you want to perform set operations like union, intersection, and checking for membership efficiently.

Example

```
# List operations
my_list1 = [1, 2, 3]
my_list2 = [3, 4, 5]
concatenated_list = my_list1 + my_list2
print(concatenated_list)  # Prints [1, 2, 3, 3, 4, 5]
```

```
# Set operations
my_set1 = {1, 2, 3}
my_set2 = {3, 4, 5}
union_set = my_set1.union(my_set2)
print(union_set)  # Prints {1, 2, 3, 4, 5}
```

In summary, lists are ordered, allow duplicates, and are mutable, while sets are unordered, do not allow duplicates, and are also mutable but with restrictions on the mutability of elements. The choice between lists and sets depends on your specific use case and requirements.

54

Difference between Sets and Dictionary

In Python, sets and dictionaries are both data structures used to store collections of items, but they serve different purposes and have distinct characteristics. Here's a differentiation between sets and dictionaries with examples:

Sets

Purpose: Sets are used to store an unordered collection of unique elements. They are primarily used for membership testing and performing set operations like union, intersection, and difference.

Structure: Sets are enclosed in curly braces {} or created using the set() constructor.

Example

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

Uniqueness: Sets do not allow duplicate elements. If you try to add a duplicate element, it will be ignored.

Access: You cannot access elements in a set by indexing because they are unordered.

Dictionaries

Purpose: Dictionaries are used to store key-value pairs, where each value is associated with a unique key. They are used for mapping and looking up values based on keys.

Structure: Dictionaries are enclosed in curly braces {} and consist of key-value pairs separated by colons:.

Example

```
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

Uniqueness: Dictionary keys must be unique, but the values can be duplicated.
Access: You can access values in a dictionary using keys.

Examples

Sets Example

```
fruits = {'apple', 'banana', 'cherry'}

# Adding an element
fruits.add('orange')

# Attempting to add a duplicate element
fruits.add('apple')  # Will not raise an error, but 'apple' won't be added again

# Removing an element
fruits.remove('banana')

# Membership testing
print('cherry' in fruits)  # Prints True

# Set operations
vegetables = {'carrot', 'broccoli', 'cherry'}
common_items = fruits.intersection(vegetables)
print(common_items)  # Prints {'cherry'}

# Iterating over a set
for fruit in fruits:
    print(fruit)
```

Dictionaries Example

```
student = {'name': 'Alice', 'age': 25, 'grade': 'A'}

# Accessing values
print(student['name'])  # Prints 'Alice'
print(student['age'])   # Prints 25

# Modifying values
student['age'] = 26

# Adding a new key-value pair
student['country'] = 'USA'
```

```
# Dictionary keys must be unique, but values can be duplicated
grades = {'Alice': 'A', 'Bob': 'B', 'Charlie': 'A', 'David': 'B'}

# Iterating over dictionary keys
for key in student:
   print(key, student[key])

# Iterating over key-value pairs using items()
for key, value in student.items():
   print(key, value)
```

In summary, sets are used to store unique elements and are suitable for set operations, while dictionaries are used to map keys to values and are suitable for looking up values based on keys. The choice between sets and dictionaries depends on the specific problem you're trying to solve in your Python program.

55

Difference between Map and Filter

In Python, map and filter are both built-in functions used for working with sequences (like lists, tuples, and iterators) to transform or filter elements based on specific criteria. However, they have different purposes and usage patterns. Here's a differentiation between map and filter with examples:

map Function

The map function is used to apply a given function to each item in an iterable (e.g., a list) and return a new iterable (typically a map object or list) containing the results of applying the function to each item. It takes two arguments: the function to apply and the iterable to apply it to.

Syntax
map(function, iterable)

Example of map:

```
# Define a function to square a number
def square(x):
   return x * x

# Apply the square function to a list of numbers
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)

# Convert the map object to a list
squared_numbers_list = list(squared_numbers)

print(squared_numbers_list)  # Prints [1, 4, 9, 16, 25]
```

filter Function

The filter function is used to filter elements from an iterable based on a given function (predicate) that returns either True or False. It returns a new iterable

(typically a filter object or list) containing the elements for which the function returned True.

Syntax

```
filter(function, iterable)
```

```
# Define a function to check if a number is even
def is_even(x):
   return x % 2 == 0

# Filter even numbers from a list
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(is_even, numbers)

# Convert the filter object to a list
even_numbers_list = list(even_numbers)

print(even_numbers_list)  # Prints [2, 4, 6]
```

Key Differences

Purpose

map is used to apply a function to each element and return a new iterable with the transformed values.

filter is used to filter elements from an iterable based on a function's criteria and return a new iterable with the filtered values.

Function Return Values

map applies a function to each element and includes the results (possibly modified values) for all elements.

filter applies a function (predicate) to each element and includes only the elements for which the function returns True.

Example Usage

Use map when you want to transform all elements of an iterable.
Use filter when you want to select specific elements from an iterable based on a condition.

Output Type

Both map and filter return iterable objects, which can be converted to lists or other iterables using `list()` or other constructors if needed.

In summary, map is for transforming elements, while filter is for selecting elements based on a condition. Both functions are powerful tools for working with iterables in Python and are often used in combination with lambda functions for conciseness.

56

Difference Between Method Overriding and Method Overloading

Method Overriding and Method Overloading are two concepts in object-oriented programming, primarily used in languages like Python and Java. They both involve defining multiple methods with the same name in a class, but they serve different purposes. Let's differentiate between them and provide examples for each:

Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. In other words, a subclass defines a method with the same name, return type, and parameters as the method in the parent class. This allows the subclass to provide its own behavior for that method while preserving the method's signature.

Key Points about Method Overriding

Occurs in inheritance when a subclass redefines a method from the parent class.
Method signature (name, return type, and parameters) must be the same in the parent and subclass.

The purpose is to provide a specialized implementation of the method in the subclass.

Example of Method overriding in Python

```
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
```

```
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Usage
dog = Dog()
print(dog.make_sound())  # Outputs "Woof!"

cat = Cat()
print(cat.make_sound())  # Outputs "Meow!"
```

In this example, both Dog and Cat classes override the make_sound method defined in the Animal class with their specific implementations.

Method Overloading

Method overloading involves defining multiple methods in a class with the same name but different parameters (i.e., a different number of parameters or different types of parameters). The choice of which method to call is determined by the number and types of arguments passed during the method invocation.

Key Points About Method Overloading

Occurs within a single class when multiple methods with the same name have different parameter lists.

The purpose is to provide multiple ways to call a method based on the arguments passed.

Python does not support method overloading in the traditional sense, as it allows only one method with a specific name in a class. However, you can achieve a similar effect by using default parameter values or variable-length argument lists (e.g., *args or **kwargs).

Example of method "overloading" using default parameters in Python:

```
class Calculator:
    def add(self, a, b=0):
        return a + b
```

```
# Usage
calc = Calculator()
result1 = calc.add(2, 3)  # Calls add(a, b) and returns 5
result2 = calc.add(2)     # Calls add(a) and returns 2

print(result1)
print(result2)
```

In this example, the add method can take one or two arguments. If only one argument is provided, it uses a default value of 0 for the second parameter.

To summarize, method overriding involves providing a specific implementation for a method in a subclass with the same name and signature, while method overloading, though not directly supported in Python, simulates providing multiple methods with the same name but different parameters within a single class.

57

Web Scrapping

Web scraping is the process of extracting data from websites. It involves sending HTTP requests to a web page, parsing the HTML content of the page, and extracting the desired information. Python provides several libraries and tools to facilitate web scraping, including requests for making HTTP requests and BeautifulSoup for parsing HTML.

Here's an explanation of the concept of web scraping with examples in Python:

1. **Installing Required Libraries:** Before you start web scraping, you need to install the necessary libraries. You can use pip to install them:

   ```
   pip install requests beautifulsoup4
   ```

2. **Sending HTTP Requests**

   You start by sending an HTTP GET request to a web page using the requests library. This retrieves the HTML content of the page.

   ```
   import requests
   url = 'https://example.com'
   response = requests.get(url)
   if response.status_code == 200:
      html_content = response.text
   else:
      print('Failed to retrieve the web page')
   ```

3. **Parsing HTML with BeautifulSoup:** Next, you use BeautifulSoup to parse the HTML content. This library helps you navigate the HTML structure and extract data.

   ```
   from bs4 import BeautifulSoup

   # Create a BeautifulSoup object
   soup = BeautifulSoup(html_content, 'html.parser')

   # Extract data using CSS selectors
   ```

```
title = soup.title.text
paragraphs = soup.find_all('p')

print('Title:', title)
print('Number of paragraphs:', len(paragraphs))
```

4. **Extracting Specific Data:** You can use BeautifulSoup to extract specific data from the HTML structure based on your requirements. For example, to extract all the links from a web page:

```
# Extract all links (anchor tags)
links = soup.find_all('a')

for link in links:
   print(link['href'])  # Print the href attribute of each link
```

5. **Handling Pagination and Pagination Links**: When scraping multiple pages, you can iterate through paginated content by following links to the next page.

```
# Example of paginated content
next_page_link = soup.find('a', text='Next')

if next_page_link:
   next_page_url = next_page_link['href']
   response = requests.get(next_page_url)
   # Continue parsing the next page's content
```

6. **Handling Dynamic Content**: Some websites load content dynamically using JavaScript. In such cases, libraries like Selenium can be used to interact with the page as a user would and scrape data.

```
from selenium import webdriver
# Set up a webdriver (e.g., Chrome)
driver = webdriver.Chrome()
driver.get(url)

# Use driver to interact with the page, wait for dynamic content, and
scrape data
```

7. **Handling Authentication**

```
# Use requests with authentication
username = 'your_username'
password = 'your_password'
auth = (username, password)
response = requests.get(url, auth=auth)
```

8. **Dealing with API**

```
# Fetch data from APIs (often returns data in JSON format)
response = requests.get(api_url)
data = response.json()
```

9. **Respect Robots.txt:** When scraping websites, it's essential to respect the site's robots.txt file, which provides guidelines on what can and cannot be scraped. You can use the robotparser library to check if a page can be scraped.

```
from urllib import robotparser
rp = robotparser.RobotFileParser()
rp.set_url('https://example.com/robots.txt')
rp.read()

if rp.can_fetch('*', url):
    # You are allowed to scrape this page
    pass
else:
    # Respect the site's rules and don't scrape
    pass
```

10. **Legal and Ethical Considerations:** When scraping websites, always ensure that you have the legal right to access and use the data. Respect the site's terms of service, use reasonable rate limiting, and avoid causing harm to the website.

Web scraping is a powerful tool for extracting data from websites, but it should be used responsibly and ethically. Always check the website's terms of use and robots.txt file before scraping, and be considerate of the site's resources and policies.

Library Name	Description	Example

BeautifulSoup	A popular library for parsing HTML and XML documents	from bs4 import BeautifulSoup
		import requests
		# Send an HTTP GET request
		url = 'https://example.com'
		response = requests.get(url)
		# Parse the HTML content of the page
		soup = BeautifulSoup(response.text, 'html.parser')
		# Extract specific data
		title = soup.title.string
		print('Title:', title)

Requests	A simple library for making HTTP requests	import requests
		# Send an HTTP GET request
		url = 'https://example.com'
		response = requests.get(url)
		# Print the content of the response
		print(response.text)

Scrapy	A powerful web scraping framework	import scrapy
		class MySpider(scrapy.Spider):
		name = 'myspider'
		start_urls = ['https://example.com']
		def parse(self, response):
		#Extract data from the response\n
		title = response.css('title::text').get()\n
		print('Title:', title)
		process= scrapy.crawler.CrawlerProcess()
		process.crawl(MySpider)
		process.start()

Selenium	A tool for automating web browsers	from selenium import webdriver\n\n# Launch a web browser\nbrowser = webdriver.Chrome()\n\n# Visit a website\nurl = 'https://example.com'\nbrowser.get(url)\n\n# Extract data using browser automation\ntitle = browser.title\nprint('Title:', title)\n\n# Close the browser\nbrowser.quit()
PyQuery	A library for jQuery-like syntax with lxml	python from pyquery import PyQuery as pq\nimport requests\n\n# Send an HTTP GET request\nurl = 'https://example.com'\nresponse =

		requests.get(url)\n\n# Parse the HTML content of the page\ndoc = pq(response.text)\n\n# Extract specific data\ntitle = doc('title').text()\nprint('Title:', title)
Goutte	A web scraping library for PHP and Symfony	python from goutte import Goutte\n\nclient = Goutte()\ncrawler = client.request('GET', 'https://example.com')\n\ndom_crawler = crawler.filter('title')\n\ntitle = dom_crawler.text()\nprint('Title:', title)
Lxml	A library for processing XML and HTML	python from lxml import html\nimport requests\n\n# Send an HTTP GET request\nurl = 'https://example.com'\nresponse = requests.get(url)\n\n# Parse the HTML content of the page\ntree = html.fromstring(response.text)\n\n# Extract specific data\ntitle = tree.xpath('//title/text()')[0]\nprint('Title:', title)
Pyppeteer	A Python port of Puppeteer, a headless Chrome	python from pyppeteer import launch\n\nasync def main():\n browser = await launch(headless=True)\n page = await browser.newPage()\n await page.goto('https://example.com')\n title = await page.title()\n print('Title:', title)\n await browser.close()\n\nimport asyncio\nasyncio.run(main())
Beautiful Soup 4k	A smaller version of BeautifulSoup (4k version)	python from bs4k import BeautifulSoup\nimport requests\n\n# Send an HTTP GET request\nurl = 'https://example.com'\nresponse = requests.get(url)\n\n# Parse the HTML content of the page\nsoup = BeautifulSoup(response.text, 'html.parser')\n\n# Extract specific data\ntitle = soup.title.string\nprint('Title:', title)

58

Introduction to NumPy

NumPy (Numerical Python) is a popular Python library used for numerical and mathematical operations. It provides support for multi-dimensional arrays and matrices, along with a wide range of mathematical functions to perform operations on these arrays efficiently. NumPy is a fundamental library for scientific computing and data analysis in Python.

Here's an explanation of NumPy with examples:

Installing NumPy
You can install NumPy using pip if it's not already installed:
pip install numpy

Importing NumPy
To use NumPy in your Python program, you need to import it:

import numpy as np

Now, let's explore some key aspects of NumPy with examples:

1. **Creating NumPy Arrays:** You can create NumPy arrays from Python lists or other iterable objects. NumPy arrays are homogeneous, meaning all elements have the same data type.

```
import numpy as np

# Creating a 1D array from a Python list
arr1d = np.array([1, 2, 3, 4, 5])

# Creating a 2D array (matrix) from a list of lists
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("1D Array:")
print(arr1d)

print("\n2D Array:")
```

```
print(arr2d)
```

2. **Array Attributes:** NumPy arrays have several useful attributes:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Shape: Returns the dimensions of the array
print("Shape:", arr.shape)

# Data Type: Returns the data type of elements in the array
print("Data Type:", arr.dtype)

# Size: Returns the total number of elements in the array
print("Size:", arr.size)

# Dimension: Returns the number of dimensions (axes)
print("Dimension:", arr.ndim)
```

3. **Array Operations:** NumPy allows you to perform various operations on arrays, such as arithmetic operations and element-wise operations.

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
result_add = arr1 + arr2

# Element-wise multiplication
result_mul = arr1 * arr2

# Dot product of two arrays
dot_product = np.dot(arr1, arr2)

print("Element-wise Addition:", result_add)
print("Element-wise Multiplication:", result_mul)
print("Dot Product:", dot_product)
```

4. **Array Indexing and Slicing:** You can access and manipulate specific elements or sub-arrays of a NumPy array using indexing and slicing.

```
import numpy as np

arr = np.array([0, 1, 2, 3, 4, 5])

# Accessing a specific element
element = arr[3]  # Retrieves the element at index 3 (0-based indexing)
print("Element at index 3:", element)

# Slicing to get a sub-array
sub_array = arr[2:5]  # Retrieves elements from index 2 (inclusive) to 5 (exclusive)
print("Sub-array:", sub_array)
```

5. **NumPy Functions:** NumPy provides a wide range of mathematical functions that can be applied to arrays efficiently.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Calculating the mean of an array
mean_value = np.mean(arr)

# Calculating the sum of array elements
sum_value = np.sum(arr)

# Finding the maximum and minimum values in an array
max_value = np.max(arr)
min_value = np.min(arr)

print("Mean:", mean_value)
print("Sum:", sum_value)
print("Max:", max_value)
print("Min:", min_value)
```

These are just some of the fundamental concepts and operations you can perform with NumPy. NumPy is extensively used in scientific computing,

machine learning, and data analysis for its efficiency and versatility when working with numerical data.

NumPy Cheat Sheet

Importing NumPy

```
import numpy as np
```

Creating Arrays

```
# 1D array from a list
arr1d = np.array([1, 2, 3, 4, 5])

# 2D array (matrix) from a list of lists
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Create an array of zeros
zeros_arr = np.zeros(3)

# Create an array of ones
ones_arr = np.ones(3)

# Create an identity matrix
identity_matrix = np.eye(3)

# Create an array with a range of values
range_arr = np.arange(0, 10, 2)  # Start, Stop, Step

# Create an array with evenly spaced values
linspace_arr = np.linspace(0, 1, 5)  # Start, End, Number of values
```

Array Attributes

```
arr = np.array([1, 2, 3, 4, 5])

# Shape: Dimensions of the array
print(arr.shape)

# Data Type: Type of elements in the array
print(arr.dtype)
```

 # Size: Total number of elements
print(arr.size)

 # Dimension: Number of dimensions (axes)
print(arr.ndim)

Array Operations

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
result_add = arr1 + arr2

# Element-wise multiplication
result_mul = arr1 * arr2

# Dot product of two arrays
dot_product = np.dot(arr1, arr2)

Array Indexing and Slicing
arr = np.array([0, 1, 2, 3, 4, 5])

# Accessing a specific element
element = arr[3]

# Slicing to get a sub-array
sub_array = arr[2:5]

NumPy Functions

arr = np.array([1, 2, 3, 4, 5])

# Mean of an array
mean_value = np.mean(arr)

# Sum of array elements
sum_value = np.sum(arr)

# Maximum and minimum values
max_value = np.max(arr)

```
min_value = np.min(arr)
```

Reshaping Arrays

```
arr = np.array([1, 2, 3, 4, 5, 6])

# Reshape into a 2x3 matrix
reshaped_arr = arr.reshape(2, 3)
```

Random Number Generation

```
# Generate random numbers
random_nums = np.random.rand(3, 3)  # Uniform distribution between 0 and 1

# Generate random integers
random_ints = np.random.randint(1, 10, size=(2, 2))  # Low, High, Size
```

Element-wise Functions

```
arr = np.array([1, 2, 3])

# Square root
sqrt_arr = np.sqrt(arr)

# Exponential
exp_arr = np.exp(arr)
```

 # Logarithm

```
log_arr = np.log(arr)
```

 # Linspace

In NumPy, linspace is a function used to create an array of evenly spaced values within a specified range. The name "linspace" stands for "linear space." It is particularly useful when you need a set of values that are evenly distributed between a start and end point, and you want to specify the number of values you want in that range.

The Basic Syntax of Linspace is as Follows

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False,
dtype=None)
```

start: The starting value of the sequence.
stop: The end value of the sequence.
num: The number of evenly spaced values you want in the array. This is an optional parameter, and the default value is 50.
endpoint: If True (default), the end value (stop) is included in the sequence. If False, it's not included.
retstep: If True, it returns a tuple with the array of values and the step size between them.
dtype: The data type of the output array (e.g., int, float). If not specified, it is inferred from the input values.
Here's an example of how to use linspace:

```
import numpy as np

# Create an array of 5 evenly spaced values between 0 and 1 (inclusive)
arr = np.linspace(0, 1, 5)

print(arr)
```

Output

```
[0. 0.25 0.5 0.75 1.  ]
```

In this example, linspace generates an array with 5 values that are evenly distributed between 0 and 1 (inclusive). The start and stop values specify the range, and the num parameter specifies the number of values. If you set endpoint=False, the end value (1.0 in this case) would not be included in the generated sequence.

linspace is particularly useful for generating data points for plotting or for creating evenly spaced values for numerical simulations and calculations.

59

Introduction to Pandas

Pandas is a popular Python library for data manipulation and analysis. It provides data structures and functions for working with structured data, making it a fundamental tool for data scientists, analysts, and engineers. In Pandas, the two primary data structures are **Series and DataFrame,** which allow you to work with one-dimensional and two-dimensional data, respectively.

Here's an explanation of Pandas with examples:

Installing Pandas

You can install Pandas using pip if it's not already installed:

```
pip install pandas
```

Importing Pandas

To use Pandas in your Python program, you need to import it:
import pandas as pd

Now, let's explore some key aspects of Pandas with examples:

1. **Series:** A Series is a one-dimensional array-like object in Pandas. It can hold data of various types and is labeled, meaning it has an index.

```
import pandas as pd

# Creating a Series from a list
data = [1, 2, 3, 4, 5]
series = pd.Series(data)

# Accessing values and index
print(series.values)
print(series.index)

# Specifying custom index
custom_index = ['A', 'B', 'C', 'D', 'E']
series = pd.Series(data, index=custom_index)
```

```
# Accessing by index label
print(series['B'])
```

2. **DataFrame:** A DataFrame is a two-dimensional tabular data structure in Pandas. It is essentially a collection of Series objects, and each Series becomes a column in the DataFrame.

```
import pandas as pd
# Creating a DataFrame from a dictionary
data =
{
        'Name': ['Alice', 'Bob', 'Charlie'],
         'Age': [25, 30, 35],
         'City': ['New York', 'San Francisco', 'Los Angeles']
}
df = pd.DataFrame(data)

# Accessing columns
print(df['Name'])
print(df['Age'])

# Accessing rows
print(df.loc[0])
```

3. **Loading Data:** Pandas provides functions to load data from various file formats, such as CSV, Excel, and SQL databases.

```
import pandas as pd

# Reading data from a CSV file
df = pd.read_csv('data.csv')

# Reading data from an Excel file
df = pd.read_excel('data.xlsx')

# Reading data from a SQL database
import sqlite3
conn = sqlite3.connect('mydb.db')
query = 'SELECT * FROM mytable'
df = pd.read_sql_query(query, conn)
```

4. **Data Exploration:** Pandas offers various functions for exploring and summarizing data.

```
# Display the first few rows of the DataFrame
print(df.head())

# Summary statistics
print(df.describe())

# Counting unique values
print(df['Category'].value_counts())

# Filtering data
filtered_df = df[df['Age'] > 25]
```

5. **Data Manipulation:** Pandas allows you to perform various data manipulation tasks, such as sorting, merging, and grouping.

```
# Sorting by a column
sorted_df = df.sort_values(by='Age', ascending=False)

# Merging DataFrames
merged_df = pd.merge(df1, df2, on='Key')

# Grouping and aggregating data
grouped_df = df.groupby('Category')['Sales'].sum()
```

6. **Data Visualization:** Pandas integrates with other libraries like Matplotlib and Seaborn for data visualization.

```
import matplotlib.pyplot as plt

# Creating a bar chart
df['Category'].value_counts().plot(kind='bar')
plt.title('Category Distribution')
plt.xlabel('Category')
plt.ylabel('Count')
plt.show()
```

These are just some of the fundamental concepts and operations you can perform with Pandas. Pandas is a versatile library for data manipulation and analysis, making it an essential tool in the data science toolkit.

60

Introduction to Matplotlib

Matplotlib is a widely used Python library for creating static, animated, and interactive visualizations in Python. It provides a flexible and comprehensive set of tools for creating a wide range of plots, charts, and graphs to help you visualize your data. In this explanation, I'll cover the basics of Matplotlib with examples.

Installing Matplotlib: You can install Matplotlib using pip if it's not already installed:

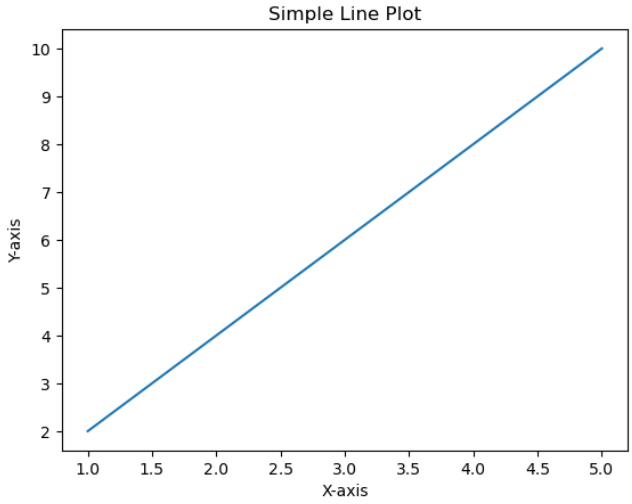
pip install matplotlib

Importing Matplotlib: To use Matplotlib in your Python program, you need to import it:

import matplotlib.pyplot as plt

Basic Plotting with Matplotlib
Let's start with a simple example of creating a basic line plot:

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Create a line plot
plt.plot(x, y)

# Adding labels and a title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')

# Display the plot
plt.show()
```



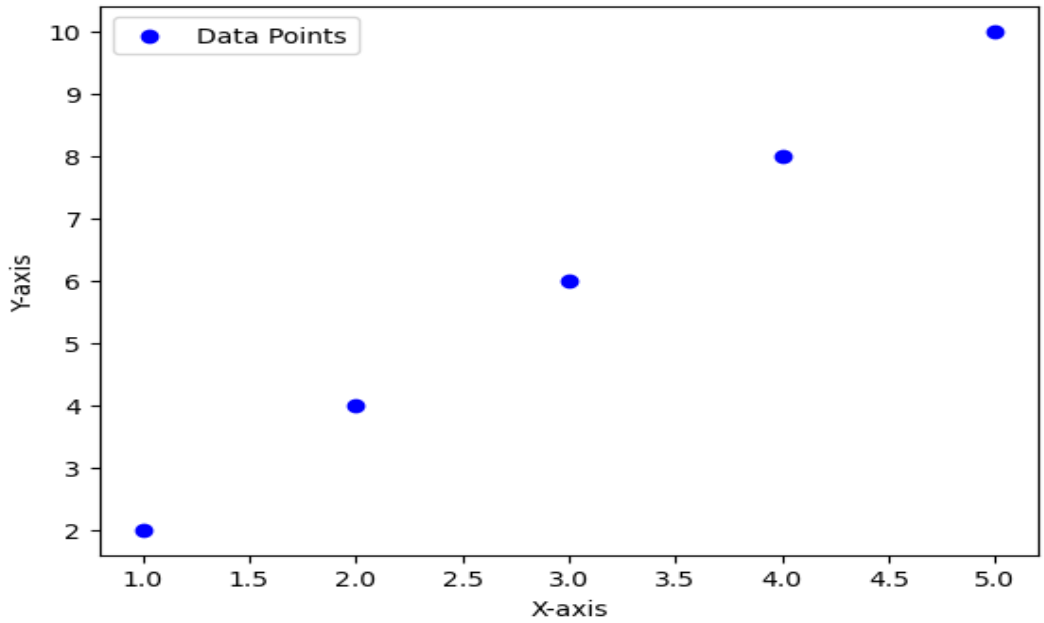
In this example:

We create a basic line plot using plt.plot(x, y).
We add labels to the X and Y axes using plt.xlabel() and plt.ylabel().
We set the title of the plot using plt.title().
Finally, we display the plot using plt.show().

Scatter Plot
Here's an example of creating a scatter plot:

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Create a scatter plot
plt.scatter(x, y, marker='o', color='blue', label='Data Points')

# Adding labels and a legend
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend()

# Display the plot
plt.show()
```

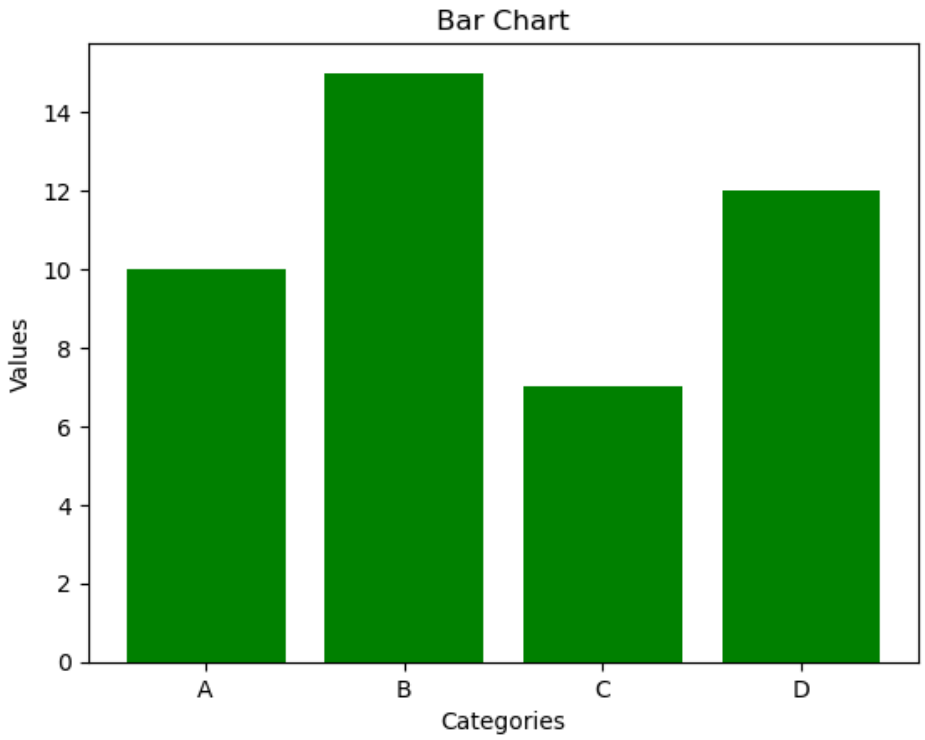
In this example, we use plt.scatter() to create a scatter plot and customize it with markers, colors, and labels.



Bar Chart: Here's an example of creating a bar chart:

```
import matplotlib.pyplot as plt
# Data
categories = ['A', 'B', 'C', 'D']
values = [10, 15, 7, 12]

# Create a bar chart
plt.bar(categories, values, color='green')

# Adding labels and a title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Chart')

# Display the plot
plt.show()
```

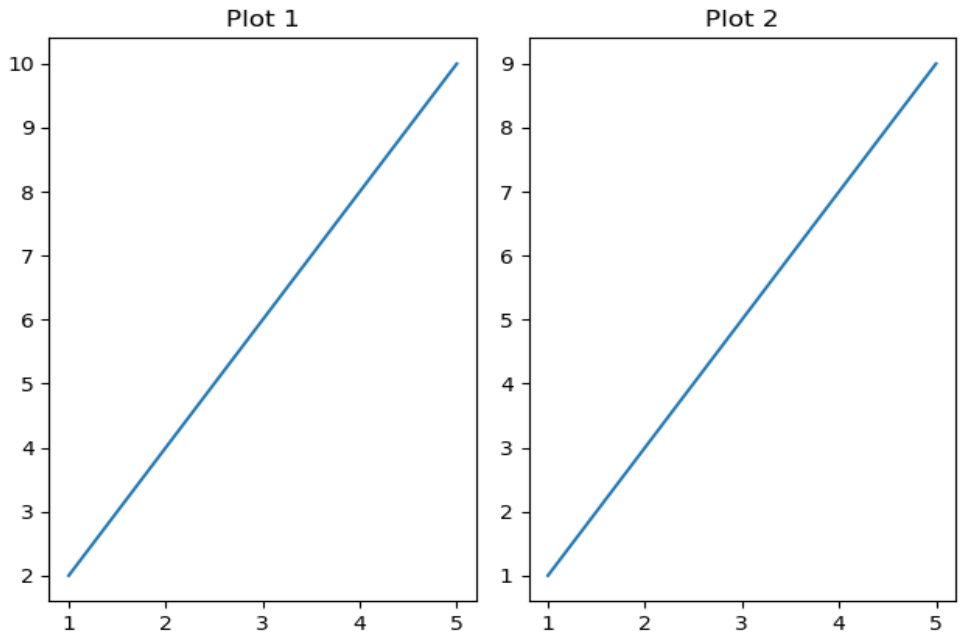
In this example, we use plt.bar() to create a bar chart and customize it with categories, values, colors, and labels.



Multiple Plots: You can create multiple plots in a single figure using subplots. Here's an example:

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [1, 3, 5, 7, 9]

# Create two subplots (1 row, 2 columns)
plt.subplot(1, 2, 1)
plt.plot(x, y1)
plt.title('Plot 1')

plt.subplot(1, 2, 2)
plt.plot(x, y2)
plt.title('Plot 2')

# Adjust layout
plt.tight_layout()

# Display the plots
plt.show()
```

In this example, we use plt.subplot() to create two subplots side by side within the same figure.



These are just some of the basic plotting examples with Matplotlib. Matplotlib provides extensive customization options and supports various types of plots, such as histograms, pie charts, and 3D plots. It is a powerful tool for data visualization and exploration in Python.

61

Introduction to Seaborn

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for creating informative and attractive statistical graphics. Seaborn is particularly useful for visualizing complex datasets with minimal effort, as it simplifies many of Matplotlib's plotting functions and adds a layer of style and aesthetics. In this explanation, I'll introduce Seaborn and provide examples of its main features.

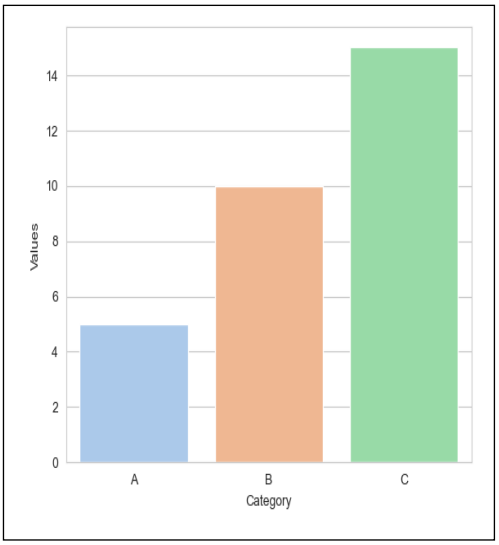
Installing Seaborn: pip install seaborn

Importing Seaborn: import seaborn as sns

Seaborn Features and Examples

1. **Styling and Themes:** Seaborn comes with different built-in themes and color palettes that can enhance the visual appearance of your plots. You can set a theme using **sns.set_style()** and choose a palette with **sns.set_palette()**. Here's an example:

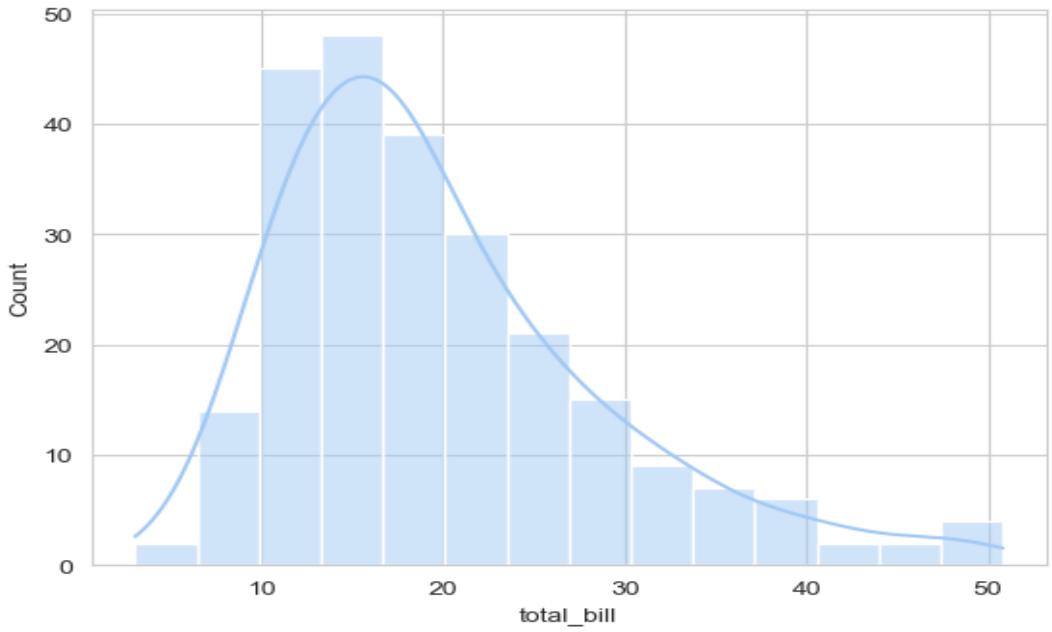
```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

# Creating a DataFrame using pandas
data = {'Category': ['A', 'B', 'C'], 'Values':
 [5, 10, 15]}
df = pd.DataFrame(data)

# Set the style and palette
sns.set_style('whitegrid')
sns.set_palette('pastel')

# Create a simple bar plot using Seaborn
sns.barplot(x='Category', y='Values', data=df)

# Show the plot
plt.show()
```



In this example, we set the style to '**whitegrid**' and the palette to '**pastel**,' and then create a bar plot using Seaborn.

2. **Distribution Plots:** Seaborn provides distribution plots like histograms, kernel density plots, and box plots to visualize the distribution of data. Here's an example of a histogram and a kernel density plot:

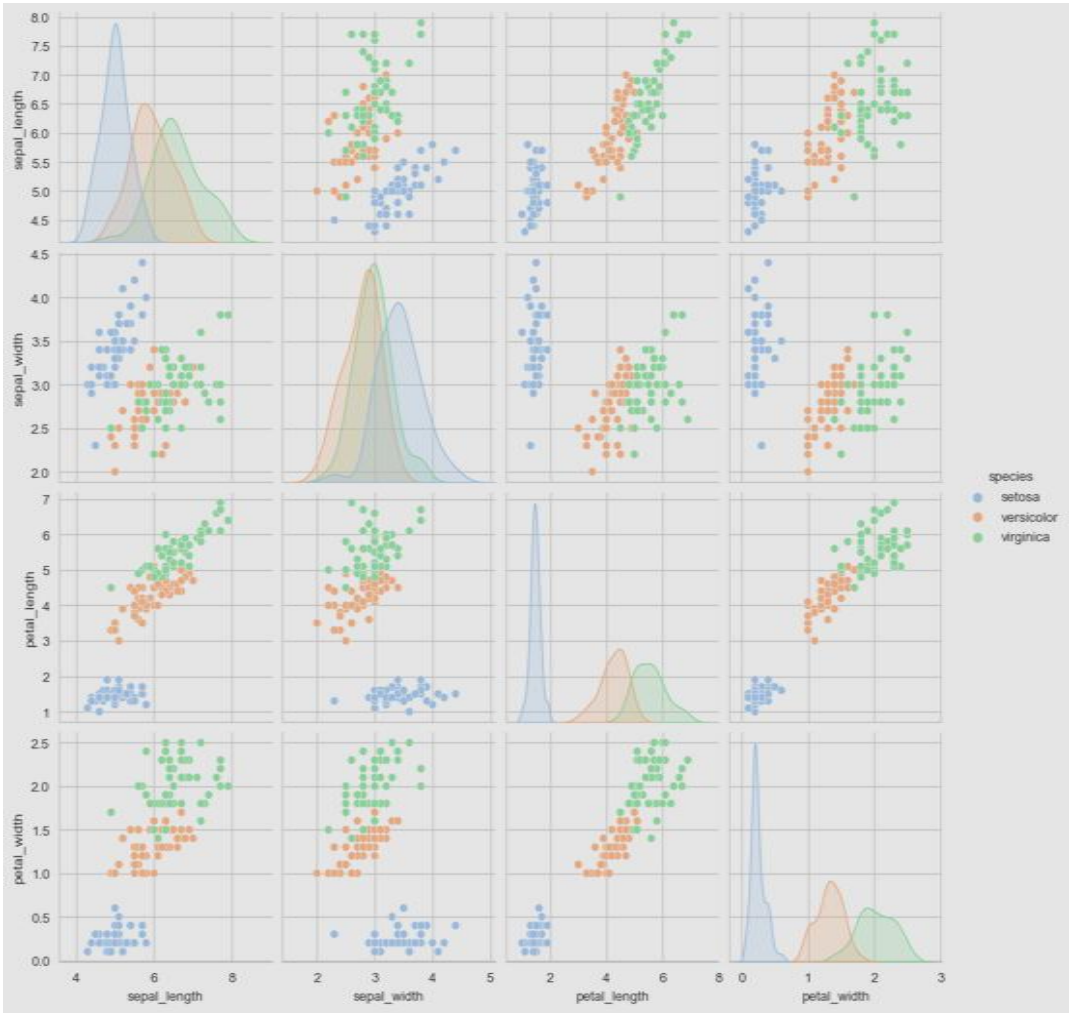
```
import seaborn as sns
import matplotlib.pyplot as plt

# Load example data
data = sns.load_dataset("tips")

# Create a histogram
sns.histplot(data["total_bill"], kde=True)

# Show the plot
plt.show()
```



In this example, we use sns.histplot() to create a histogram with a kernel density plot overlay.

3. **Pair Plots:** Pair plots are a great way to visualize pairwise relationships between variables in a dataset. Seaborn's pairplot function automatically creates scatterplots for numerical variables and histograms for numerical and categorical variables.

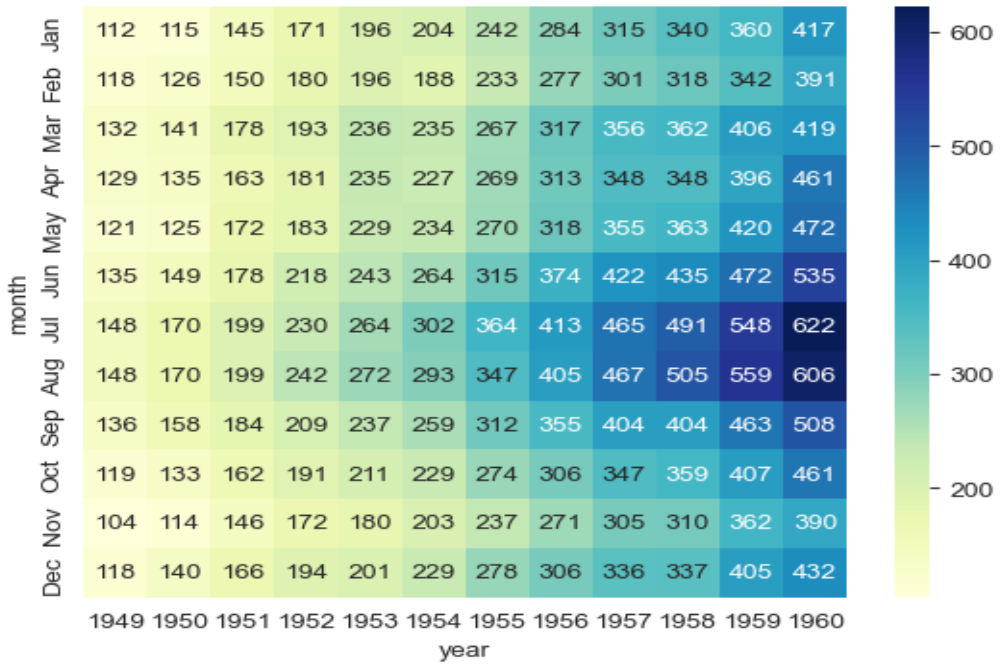
```
import seaborn as sns
import matplotlib.pyplot as plt

# Load example data
data = sns.load_dataset("iris")

# Create a pair plot
sns.pairplot(data, hue="species")
# Show the plot
plt.show()
```



In this example, we use sns.pairplot() to create a pair plot for the Iris dataset, coloring the points by species.

4. **Heatmaps:** Heatmaps are useful for visualizing the correlation matrix of a dataset. Seaborn's heatmap function can create heatmaps with annotated values.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load example data
data = sns.load_dataset("flights")
pivot_data = data.pivot_table(index="month", columns="year",
values="passengers")

# Create a heatmap
sns.heatmap(pivot_data, cmap="YlGnBu", annot=True, fmt="d")

# Show the plot
plt.show()
```

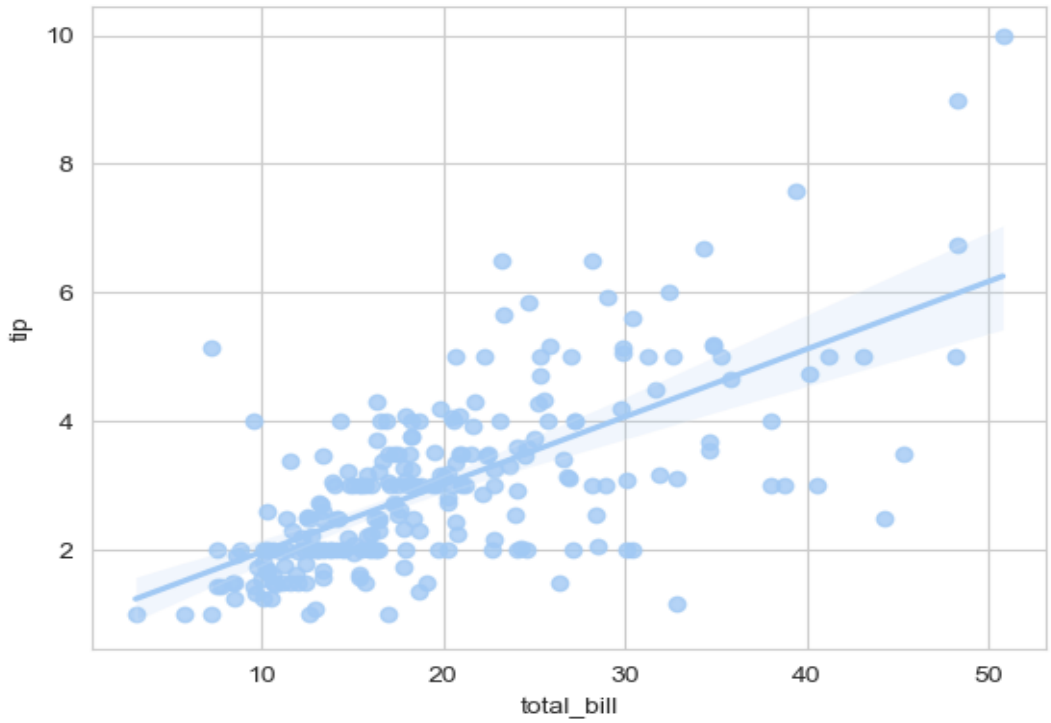


In this example, we use sns.heatmap() to create a heatmap of the passenger counts for different months and years.

5. **Regression Plots:** Seaborn makes it easy to create regression plots for visualizing relationships between two variables and fitting regression models.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load example data
```

```
data = sns.load_dataset("tips")

# Create a regression plot
sns.regplot(x="total_bill", y="tip", data=data)

# Show the plot
plt.show()
```



In this example, we use sns.regplot() to create a regression plot between total bill and tip amounts.

These are just a few examples of what you can do with Seaborn. It offers a wide range of visualization options and customization capabilities, making it a valuable tool for data exploration and presentation in Python.

62

Introduction to Tkinter

Tkinter is a standard Python library for creating graphical user interfaces (GUIs). It provides a set of tools and widgets for building windows, dialog boxes, buttons, labels, textboxes, and more. With Tkinter, you can create desktop applications with a graphical interface. In this explanation, I'll introduce the basic concepts of Tkinter and provide examples of creating a simple GUI application.

Importing Tkinter: To use Tkinter, you need to import it:

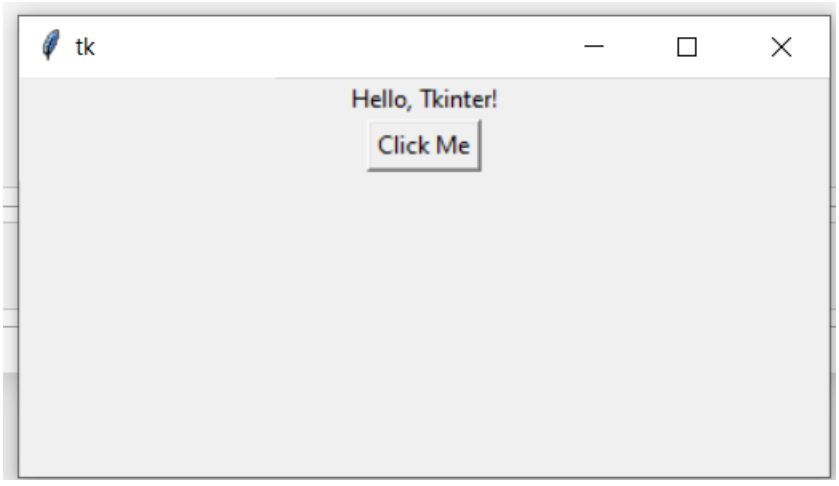
```
import tkinter as tk
```

Creating a Basic Tkinter Application: Here's an example of a minimal Tkinter application that creates a simple window:

```
import tkinter as tk
# Create a main window
window = tk.Tk()
# Add a label to the window
label = tk.Label(window, text="Hello, Tkinter!")
label.pack()
# Start the main loop
window.mainloop()
```

Adding Buttons and Event Handling: Let's enhance the previous example by adding a button and an event handler:

```
import tkinter as tk

# Function to handle button click
def on_button_click():
   label.config(text="Button Clicked!")

# Create a main window
window = tk.Tk()

# Add a label to the window
label = tk.Label(window, text="Hello, Tkinter!")
label.pack()

# Add a button to the window
button = tk.Button(window, text="Click Me", command=on_button_click)
button.pack()

# Start the main loop
window.mainloop()
```



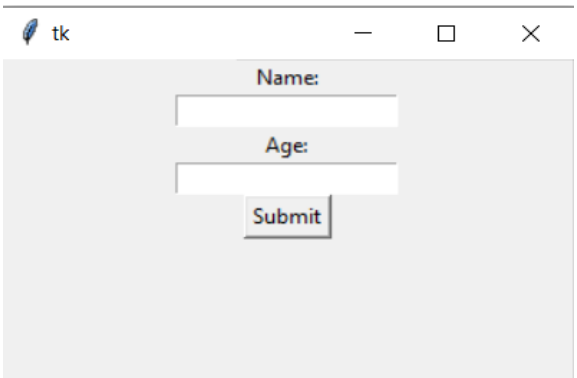
In this example

We define a function **on_button_click()** to be called when the button is clicked. We create a button widget using **tk.Button()** and associate the **on_button_click** function with the command parameter.

When the button is clicked, the **on_button_click** function changes the text of the label.

Creating a Simple Form: Let's create a simple form with labels, entry widgets, and a submit button:

```
import tkinter as tk
# Function to handle form submission
def submit_form():
    name = name_entry.get()
    age = age_entry.get()
    result_label.config(text=f"Name: {name}, Age: {age}")

# Create a main window
window = tk.Tk()

# Add labels, entry widgets, and a button
tk.Label(window, text="Name:").pack()
name_entry = tk.Entry(window)
name_entry.pack()

tk.Label(window, text="Age:").pack()
age_entry = tk.Entry(window)
age_entry.pack()

submit_button = tk.Button(window, text="Submit", command=submit_form)
submit_button.pack()

result_label = tk.Label(window, text="")
result_label.pack()
# Start the main loop
window.mainloop()
```



In this example

We create labels, entry widgets, and a button to create a simple form.The submit_form() function retrieves the values from the entry widgets when the button is clicked and updates the result label with the information.

These examples provide a basic introduction to creating GUI applications with Tkinter in Python. Tkinter offers a wide range of widgets and features for building more complex and interactive desktop applications.

63

Python Tips

Here's a list of Python tips to help you become a more efficient and effective Python programmer:

Use Descriptive Variable Names: Choose meaningful names for your variables and functions to enhance code readability.

Follow PEP 8: Adhere to the Python Enhancement Proposal (PEP) 8 style guide for consistent code formatting.

Use a Virtual Environment: Isolate project dependencies using virtual environments to prevent conflicts.

Leverage List Comprehensions: Simplify list creation and manipulation with list comprehensions.

Avoid Global Variables: Minimize the use of global variables to reduce complexity and improve code maintainability.

Keep Functions Short and Focused: Aim for small, focused functions that perform a single task.

Document Your Code: Use docstrings and comments to explain your code's purpose and usage.

Learn Built-in Functions: Familiarize yourself with Python's built-in functions to avoid reinventing the wheel.

Use Iterators and Generators: Utilize iterators and generators to process large datasets efficiently.

Handle Exceptions Gracefully: Use try-except blocks to handle exceptions and errors gracefully.

Use Context Managers: Employ context managers (e.g., **with** statements) for resource management.

Optimize Loops: Try to optimize loops by moving constant calculations outside the loop when possible.

Profile Your Code: Use profilers to identify performance bottlenecks in your code.

Choose the Right Data Structure: Select the appropriate data structure (lists, sets, dictionaries, etc.) for your task.

Know Built-in Types: Familiarize yourself with Python's built-in data types and their methods.

Master String Formatting: Learn the various ways to format strings, including f-strings, % formatting, and str.format().

Understand Mutable vs. Immutable: Understand the distinction between mutable (e.g., lists) and immutable (e.g., tuples) data types.

Keep Your Dependencies Updated: Regularly update your project's dependencies to benefit from bug fixes and new features.

Use Virtual Environments: Isolate project dependencies using virtual environments to avoid conflicts.

Write Unit Tests: Implement unit tests to ensure your code behaves as expected and catches regressions.

Learn List Slicing: Master list slicing to efficiently manipulate and extract data from lists.

Use the enumerate() Function: Iterate over both index and value with enumerate() in loops.

Avoid Hardcoding Values: Avoid hardcoding constants; use named constants or configuration files.

Utilize the collections Module: Explore the collections module for specialized data structures like defaultdict and Counter.

Practice DRY (Don't Repeat Yourself): Reuse code through functions or classes rather than duplicating it.

Learn Regular Expressions: Familiarize yourself with regular expressions for powerful text processing.

Handle Time and Dates with datetime: Use the datetime module for working with dates and times.

Optimize Imports: Import only what you need to reduce clutter and improve readability.

Use List and Dictionary Comprehensions: Use comprehensions to create and manipulate lists and dictionaries efficiently.

Explore Functional Programming: Learn functional programming concepts like map, filter, and reduce.

Understand Closures: Understand how closures work in Python and how they can be useful.

Learn Decorators: Explore decorators for modifying or extending the behavior of functions.

Avoid Using Global: Limit the use of the global keyword; prefer passing variables as arguments.

Use the logging Module: Implement logging for better debugging and monitoring.

Profile Your Code: Use profiling tools to identify performance bottlenecks and optimize your code.

Handle Exceptions Gracefully: Use try and except to handle errors and exceptions effectively.

Explore Third-party Libraries: Discover and utilize third-party libraries to extend Python's capabilities.

Learn List Methods: Familiarize yourself with list methods like append(), extend(), and sort().

Use Generators for Large Datasets: Use generators to process large datasets efficiently without loading them entirely into memory.

Master Object-Oriented Programming (OOP): Understand OOP principles and how to create classes and objects.

Customize Exception Handling: Create custom exceptions to provide meaningful error messages.

Use the zip() Function: Combine multiple iterable elements with the zip() function.

Know the collections Module: Explore advanced data structures like namedtuple, deque, and ChainMap in the collections module.

Use Set Operations: Leverage set operations (union, intersection, etc.) for efficient set manipulation.

Optimize with List Comprehensions: Filter and transform data efficiently using list comprehensions.

Apply Functional Programming: Implement functional programming concepts like `map()`, `filter()`, and `reduce()`.

Learn Context Managers: Understand and create custom context managers for resource management.

Use Type Hinting: Employ type hinting for improved code documentation and static analysis.

Write Unit Tests: Develop unit tests to ensure code correctness and maintainability.

Keep Learning: Python is a vast language with a rich ecosystem. Keep learning and exploring new features and libraries to become a better Python programmer.

Avoid Using `eval()`: Refrain from using `eval()` to execute arbitrary code, as it can pose security risks.

Consider Using with for Files: Use the with statement (with open(...) as ...) when working with files to ensure proper resource management.

Master Dictionaries: Become proficient in working with dictionaries, as they are versatile and widely used in Python.

Use sorted() with Custom Key Functions: Sort lists and other iterables with sorted() using custom key functions for complex sorting criteria.

Leverage Python's Ecosystem: Explore and utilize Python's vast ecosystem of libraries and frameworks to streamline development.

Use List Comprehensions for Filtering: Use list comprehensions with conditional expressions to filter elements efficiently.

Customize print(): Customize the print() function with optional parameters like sep and end for more control over output formatting.

Understand __init__.py Files: Learn how to use __init__.py files to make directories act as Python packages.

Avoid Mutating Lists During Iteration: Avoid modifying a list while iterating over it to prevent unexpected behavior.

Experiment and Refactor: Don't hesitate to experiment with different approaches, and refactor your code for continuous improvement.

Use is for Identity Comparison: Use **is** to compare object identity (memory address) and == for value equality.

Apply Defaultdict for Default Values: Utilize collections.defaultdict to provide default values for dictionary keys.

Use the heapq Module for Heap Operations: Explore the **heapq** module for efficient heap operations like creating heaps and heapsorting.

Handle JSON Data with json Module: Read and write JSON data easily using Python's built-in json module.

Learn and Use List Methods: Familiarize yourself with list methods like append(), extend(), insert(), and pop() for list manipulation.

Use the math Module for Mathematical Operations: The math module offers a wide range of mathematical functions and constants for numerical tasks.

Work with Dates and Times Using Datetime: Utilize the datetime module to perform date and time calculations and formatting.

Explore Regular Expressions: Master regular expressions (re module) for advanced text search and manipulation.

Optimize String Concatenation: Use str.join() for efficient string concatenation instead of repeated + operations.

Be Mindful of Mutable Default Arguments: Be cautious when using mutable objects like lists as default function arguments.

Learn and Use Decorators: Explore decorators for modifying or extending the behavior of functions and methods.

Understand Generators and Yield: Create generators using the yield keyword to efficiently generate and process data on-the-fly.

Implement __str__ and __repr__ Methods: Define these special methods in your classes for customized string representations.

Learn about Python's Garbage Collection: Understand how Python's garbage collection works to manage memory efficiently.

Optimize Imports: Organize and optimize your imports to make your code more readable and maintainable.

Use try...else...finally Blocks: Leverage the else and finally blocks in exception handling for better control and cleanup.

Dive into Object-Oriented Programming (OOP): Understand OOP principles like inheritance, encapsulation, and polymorphism.

Work with File Paths Using os.path: Use the os.path module for platform-independent file path manipulation.

Leverage Closures for Encapsulation: Use closures to encapsulate data and functions within a single scope.

Learn and Use collections.deque: Use collections.deque for efficient and thread-safe double-ended queues (dequeues).

Know When to Use map() and filter(): Learn when to use map() and filter() for efficient iterable processing.

Use itertools for Advanced Iteration: Explore the itertools module for advanced tools for creating iterators.

Understand and Use Python's asyncio: Dive into asynchronous programming with Python's asyncio library for concurrent and non-blocking code execution.

Optimize Recursive Functions: Implement memoization or tail recursion for more efficient recursive functions.

Master Context Managers: Create custom context managers using the contextlib module for resource management and clean code.

Avoid Using * for Importing Everything: Avoid using wildcard imports (e.g., from module import *) to maintain code clarity.

Use the bytes and bytearray Types for Binary Data: The bytes and bytearray types are ideal for handling binary data.

Keep a Clean and Tidy Codebase: Regularly refactor and clean up your codebase to improve maintainability.

Learn and Use functools.partial: Customize functions by using functools.partial to fix certain arguments.

Know How to Extend Classes: Understand how to extend built-in classes and create your own custom classes.

Use List and Dictionary Unpacking: Unpack lists and dictionaries elegantly using unpacking syntax (* and **).

Write Docstrings for Functions and Modules: Document your code with docstrings for clear explanations and automated documentation generation.

Explore Web Development with Python: Learn web development frameworks like Django or Flask for building web applications.

Keep Up with Python Updates: Stay informed about new Python versions and their features and migrate to newer versions as appropriate.

Consider Testing Frameworks: Explore testing frameworks like pytest for comprehensive test suites.

Use collections.namedtuple for Simple Classes: Simplify class creation with collections.namedtuple for classes with a fixed set of attributes.

Learn About Python's Global Interpreter Lock (GIL): Understand the implications of the Global Interpreter Lock on multi-threaded Python programs.

Handle Command-Line Arguments with argparse: Utilize the argparse module for parsing command-line arguments in a structured way.

Optimize with functools.lru_cache: Speed up functions with expensive calculations by using functools.lru_cache for caching results.

Experiment with Jupyter Notebooks: Explore Jupyter notebooks for interactive Python coding, data exploration, and visualization.

Utilize Python Data Serialization Libraries: Learn about libraries like Pickle and JSON for serializing and deserializing data.

Understand the GIL's Impact on Multi-Threading: Be aware of the Global Interpreter Lock (GIL) when working with multi-threading in Python.

Work with CSV and Excel Data: Familiarize yourself with libraries like csv and pandas for handling CSV and Excel data.

Create Python Scripts for Automation: Write Python scripts to automate repetitive tasks and save time.

Use unittest.mock for Testing: Employ the unittest.mock module to create and manipulate mock objects for testing.

Know How to Profile Code: Learn how to use profiling tools like cProfile to identify performance bottlenecks.

Consider Using Python for Data Science: Explore Python libraries like NumPy, pandas, and scikit-learn for data analysis and machine learning.

Understand the Global State in Modules: Be cautious about global variables and state in modules; prefer passing data explicitly.

Learn About Python Design Patterns: Study and apply common design patterns to write clean and maintainable code.

Stay Active in the Python Community: Participate in Python forums, meetups, and conferences to learn from others and share your knowledge.

Continuing to practice and apply these tips will help you become a more proficient Python programmer and unlock the full potential of the language for various applications.

These Python tips cover a wide range of topics and can help you write more efficient, readable, and maintainable Python code.

64

Python Tricks

Sl. No	Python Tricks	Example
1	Flatten the lists	import itertools a = [[10, 20], [30, 40], [50, 60]] b = list(itertools.chain.from_iterable(a)) print(b) Output: [10, 20, 30, 40, 50, 60]
2	Reverse a list	a = ["10", "9", "8", "7"] print(a[::-1]) Output: 10, 9, 8, 7
3	Combining different lists	a = ['a', 'b', 'c', 'd'] b = ['e', 'f', 'g', 'h'] for x, y in zip(a, b): print(x, y) Output: a e, b f, c g, d h
4	Negative indexing lists	a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] a[-3:-1] Output: [8, 9]
5	Analyzing the most frequent on the list	a = [1, 2, 3, 4, 2, 2, 3, 1, 4, 4, 4] print(max(set(a), key=a.count)) Output: 4
6	Reversing the string	a = "python" print("Reverse is", a[::-1]) Output: Reverse is nohtyp
7	Splitting the string	a = "Python is the language of the future" b = a.split() print(b) Output: ['Python', 'is', 'the', 'language', 'of', 'the', 'future']
8	Printing out multiple values of strings	print("on" * 3 + ' ' + "off" * 2) Output: ononon offoff
9	Creating a single string	a = ["I", "am", "not", "available"] print(" ".join(a)) Output: I am not available
10	Checking if two words are anagrams	from collections import Counter def is_anagram(str1, str2): return Counter(str1) == Counter(str2) print(is_anagram('taste', 'state')) print(is_anagram('beach', 'peach')) Output: True, False

11	Transposing a matrix	mat = [[8, 9, 10], [11, 12, 13]] new_mat = zip(*mat) for row in new_mat: print(row) Output: (8, 11), (9, 12), (10, 13)
12	Chaining comparison operators	a = 17 b = 21 c = 11 print(c < a) print(a < b) Output: True, True, True
13	Inverting the Dictionary	dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7} dict2 = {v: k for k, v in dict1.items()} print(dict2) Output: {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 6: 'f', 7: 'g'}
14	Iterating value pairs and dictionary keys	dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6} for a, b in dict1.iteritems(): print('{: {}'.format(a, b)) Output: a: 1, b: 2, c: 3, d: 4, f: 6
15	Merging multiple dictionaries	x = {'a': 1, 'b': 2} y = {'b': 3, 'c': 4} z = {**x, **y} print(z) Output: {'a': 1, 'b': 3, 'c': 4}
6	Initializing empty spaces	a_list = list() a_dict = dict() a_map = map() a_set = set()
17	Initializing lists filled with numbers	listA = [1] * 1000 listB = [2] * 1000
18	Checking and analyzing the memory unit of an object	import sys a = 10 print(sys.getsizeof(a)) Output: 28
19	Swapping values	x, y = 13, 26 x, y = y, x print(x, y) Output: 26 13
20	Implementing the map function	In competitive coding, you might come across an input like this: 1234567890 To get the input as a list of numbers, perform the following: list(map(int, input().split())) Note: Always use the input() function irrespective of the type of input and convert it using the map function.
21	Merging different lists	The **Collections** module allows you to remove duplicates from a list. In Java, you have to use the **HashMap** to remove duplicate modules, but it's far easier in the case of Python. **print(list(set([1, 2, 3, 4, 3, 4, 5, 6, 7, 8, 9])))** Output: **[1, 2, 3, 4, 5, 6, 7, 8, 9]** You need to use **extend()** and **append()** in the lists while merging multiple lists. **a = [1, 2, 3, 4]** **b = [5, 6, 7, 8,**

		9] Note: **a.extend(b)** will display one list. **a** **[1, 2, 3, 4]** Note: **a.append(b)** will display the list of lists. **a** **[1, 2, 3, 4, [5, 6, 7, 8, 9]]**
22	Writing code within functions	In Python, it is always better to write your code within functions. **def main():** **for i in range(2 ** 3):** **print(x)** **main()** The above code fragment is better than the one below: **for x in range(2 ** 3):** **print(x)** The CPython implementation saves time in the case of storing local variables.
23	Bonus tip	These useful Python tricks will help you code better and more efficiently. Here is a bonus tip that you should know and implement. Strings concatenation: **str1 = ""** **some_list = ["Welcome ", "To ", "Bonus ", "Tips "]** **print(str1.join(some_list))** Use the above code instead of: **str1 = ""** **some_list = ["Welcome ", "To ", "Bonus ", "Tips "]** **for x in some_list:** **str1 += x** **print(str1)** Speed and, most of all, efficiency, are key to coding better. By incorporating the tips outlined in this article, you can significantly improve your Python programming skills. Give them a try in your next competitive coding event or other Python projects and notice the difference they make.
24	Use Namedtuples for Readability	Namedtuples are a convenient way to define simple classes for storing data records. They provide both named fields and immutability. Example: from collections import namedtuple Point = namedtuple('Point', ['x', 'y']) p = Point(1, 2) print(p.x, p.y) Output: 1 2
25	Understand and Use List Comprehensions	List comprehension is a concise way to create lists. Example: squares = [x**2 for x in range(10)] print(squares) Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
26	Master Python's enumerate Function	enumerate allows you to loop over iterable objects while keeping track of the current index. Example: fruits = ['apple', 'banana', 'cherry'] for idx, fruit in enumerate(fruits): print(idx, fruit)

27	Swap Values Using Tuple Packing/Unpacking	You can swap values of variables without using a temporary variable using tuple packing/unpacking. Example: a, b = 1, 2 a, b = b, a print(a, b) Output: 2 1
28	Use the any and all Functions for Conditions	any checks if at least one element in an iterable is True, while all checks if all elements are True. Example: numbers = [True, False, True] print(any(numbers)) print(all(numbers)) Output: True (at least one True), False (not all are True)

29	Learn About Python's Generators	Generators allow you to create iterators without building large data structures in memory. Example:
		def square_numbers(n):
		 for i in range(n):
		 yield i**2
		for num in square_numbers(5):
		 print(num)

30	Use List Slicing for Sublists	List slicing lets you create sublists efficiently. Example: my_list = [1, 2, 3, 4, 5] sublist = my_list[1:4] print(sublist) Output: [2, 3, 4]
31	Explore Python's zip Function for Iteration	zip combines multiple iterables element-wise. Example: names = ['Alice', 'Bob', 'Charlie'] scores = [85, 92, 78] for name, score in zip(names, scores): print(name, score)
32	Utilize Python's sorted Function	sorted allows you to sort iterables in ascending order. Example: numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5] sorted_numbers = sorted(numbers) print(sorted_numbers) Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
33	Use Python's filter Function for Filtering	filter filters elements from an iterable based on a function. Example: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] even_numbers = list(filter(lambda x: x % 2 == 0, numbers)) print(even_numbers) Output: [2, 4, 6, 8, 10

65

Sample Python Interview Questions and Answers

1. **What is Python?**

   Answer: Python is a high-level, interpreted programming language known for its simplicity and readability. It is widely used for web development, data analysis, machine learning, and more.

2. **What are the key features of Python?**

   Answer: Key features of Python include easy-to-read syntax, dynamic typing, automatic memory management, and extensive standard libraries.

3. **How is Python different from other programming languages like Java or C++?**

   Answer: Python is dynamically typed, has automatic memory management, and emphasizes simplicity and readability, making it different from statically-typed languages like Java and C++.

4. **Explain Python's GIL (Global Interpreter Lock).**

   Answer: The GIL is a mutex in Python that allows only one thread to execute in the interpreter at a time. It can limit the multi-threading performance in CPU-bound applications.

5. **What are the different data types in Python?**

   Answer: Python has various built-in data types, including int, float, str, list, tuple, set, and dict.

6. **What is a Python list comprehension?**

   Answer: List comprehensions provide a concise way to create lists. For example, [x for x in range(10)] generates a list of numbers from 0 to 9.

7. **Explain the difference between 'deep copy' and 'shallow copy' in Python.**

   Answer: A shallow copy creates a new object, but it doesn't create copies of nested objects. A deep copy creates new objects for both the outer object and all nested objects.

8. **What is the purpose of the 'if name == "main":' statement in Python scripts?**

   Answer: It allows you to check whether the script is being run as the main program or imported as a module. Code within this block only executes if it's the main program.

9. **What is a Python generator, and how is it different from a list?**

   Answer: A generator is an iterable that produces values on-the-fly using a yield statement. Unlike lists, generators do not store all values in memory simultaneously, making them memory-efficient.

10. **Explain the purpose of Python's 'enumerate' function.**

    Answer: The enumerate function is used to iterate through an iterable while keeping track of the index or position of the current item.

11. **How do you handle exceptions in Python?**

    Answer: You can use try, except, and optionally finally blocks to handle exceptions. For example:

```
try:
    # Code that may raise an exception
except SomeException as e:
    # Handle the exception
else:
    # Code to run if no exception is raised
finally:
    # Code that always runs
```

12. What are decorators in Python?

Answer: Decorators are a way to modify or extend the behavior of functions or methods without changing their source code. They are often used for aspects such as logging, authentication, or memoization.

13. Explain the use of 'lambda' functions in Python.

Answer: Lambda functions are small, anonymous functions defined using the lambda keyword. They are typically used for simple operations and can be passed as arguments to higher-order functions.

14. What is the difference between 'deep copy' and 'shallow copy' in Python?

Answer: A shallow copy creates a new object but doesn't create copies of nested objects. A deep copy creates new objects for both the outer object and all nested objects, recursively.

15. What is a Python package?

Answer: A package is a way of organizing related Python modules into a single directory hierarchy. It helps in modularizing and structuring larger Python applications.

16. Explain the purpose of Python's 'with' statement.

Answer: The with statement is used to simplify resource management (e.g., file handling). It ensures that cleanup code is executed even if an exception occurs.

17. What is the Global Interpreter Lock (GIL) in Python, and how does it affect multi-threaded programs?

Answer: The GIL is a mutex that protects access to Python objects. It allows only one thread to execute Python code at a time. In multi-threaded CPU-bound programs, it can limit performance, but it doesn't affect I/O-bound or multi-processing tasks.

18. Explain the purpose of the 'yield' keyword in Python.

Answer: The yield keyword is used in a function to turn it into a generator. It allows the function to generate a value and then pause its execution,

allowing for efficient memory usage in situations where the result set could be large.

19. **What is the difference between 'append' and 'extend' methods in Python lists?**

    Answer: The append method adds its entire argument as a single element to the end of a list. The extend method adds each element of its argument to the list.

20. **What is a Python virtual environment, and why is it useful?**

    Answer: A virtual environment is an isolated Python environment where you can install packages independently of the system-wide Python installation. It helps manage dependencies and avoids conflicts between packages in different projects.

21. **What is the purpose of the 'self' keyword in Python classes?**

    Answer: In Python, self refers to the instance of the class itself and is used to access instance variables and methods within the class.

22. **Explain the use of 'args' and 'kwargs' in Python function definitions.**

    Answer: *args and **kwargs allow functions to accept a variable number of positional and keyword arguments, respectively. *args collects extra positional arguments into a tuple, while **kwargs collects extra keyword arguments into a dictionary.

23. **What is a Python dictionary comprehension?**

    Answer: Dictionary comprehensions provide a concise way to create dictionaries. For example, {k: k**2 for k in range(5)} generates a dictionary with squares of numbers as key-value pairs.

24. **What is a Python set, and what is its main use case?**

    Answer: A set is an unordered collection of unique elements. Its primary use case is to eliminate duplicate values from a sequence.

25. Explain the purpose of the 'super()' function in Python classes.

Answer: super() is used to call a method from a parent class. It is often used to extend the behavior of a method in a child class.

26. What is the difference between 'deep copy' and 'shallow copy' in Python?

Answer: A shallow copy creates a new object, but it doesn't create copies of nested objects. A deep copy creates new objects for both the outer object and all nested objects.

27. Explain the Global Interpreter Lock (GIL) in Python.

Answer: The Global Interpreter Lock (GIL) is a mutex in CPython (the most commonly used implementation of Python) that allows only one thread to execute Python bytecode at a time. This can limit multi-threading performance in CPU-bound applications.

28. What is the purpose of the 'with' statement in Python?

Answer: The with statement is used for context management. It ensures that resources are acquired and released properly, such as in file handling (auto-closing files).

29. What is a Python generator, and how is it different from a list?

Answer: A generator is an iterable that generates values on-the-fly using a yield statement. Unlike lists, generators do not store all values in memory simultaneously, making them memory-efficient.

30. Explain the purpose of Python's 'enumerate' function.

Answer: The enumerate function is used to iterate through an iterable while keeping track of the index or position of the current item.

31. What is a Python decorator, and how is it used?

Answer: A decorator is a function that takes another function and extends or modifies its behavior without changing its source code. Decorators are often used for aspects like logging, authentication, or memoization.

32. How can you handle exceptions in Python?

Answer: You can use try, except, and optionally finally blocks to handle exceptions. For example:

```
try:
    # Code that may raise an exception
except SomeException as e:
    # Handle the exception
else:
    # Code to run if no exception is raised
finally:
    # Code that always runs
```

33. What is the purpose of 'lambda' functions in Python?

Answer: Lambda functions are small, anonymous functions defined using the lambda keyword. They are typically used for simple operations and can be passed as arguments to higher-order functions.

34. What is the difference between 'deep copy' and 'shallow copy' in Python?

Answer: A shallow copy creates a new object, but it doesn't create copies of nested objects. A deep copy creates new objects for both the outer object and all nested objects, recursively.

35. What is a Python package, and how is it different from a module?

Answer: A package is a way of organizing related Python modules into a single directory hierarchy. Packages contain multiple modules, while a module is a single Python file.

36. Explain the purpose of Python's 'yield' keyword.

Answer: The yield keyword is used in a function to turn it into a generator. It allows the function to generate a value and then pause its execution, allowing for efficient memory usage in situations where the result set could be large.

37. What is the difference between 'append' and 'extend' methods in Python lists?

Answer: The append method adds its entire argument as a single element to the end of a list. The extend method adds each element of its argument to the list.

38. What is a Python virtual environment, and why is it useful?

Answer: A virtual environment is an isolated Python environment where you can install packages independently of the system-wide Python installation. It helps manage dependencies and avoids conflicts between packages in different projects.

39. How do you comment on multiple lines in Python?

Answer: You can use triple quotes (either single or double) to comment multiple lines, although this is typically used as docstrings for functions and classes.

40. What is the purpose of the 'pass' statement in Python?

Answer: The pass statement is a no-op statement that does nothing. It is often used as a placeholder to avoid syntax errors while writing code.

66

Top 100 Python Interview Questions

1. What is Python?
2. What are the benefits of using Python language in the present scenario?
3. Is Python a compiled language or an interpreted language?
4. What is the significance of the ‘#’ symbol in Python?
5. How would you describe Python as a dynamically typed language?
6. Explain the difference between Mutable and Immutable data types in Python.
7. How are arguments passed in Python, by value or by reference?
8. What distinguishes a Set from a Dictionary in Python?
9. What are *args and *kwargs in Python functions?
10. Is indentation mandatory in Python code?
11. Define the concept of Scope in Python.
12. What is a docstring in Python, and why is it used?
13. Explain the concept of a namespace in Python.
14. What are Access Specifiers in Python?
15. In Python, what are the roles of 'break,' 'continue,' and 'pass'?
16. Differentiate between 'for' loops and 'while' loops in Python.
17. Can you pass a function as an argument in Python?
18. What's the difference between '/' and '//' in Python division?
19. How do you implement exceptional handling in Python?
20. What is List Comprehension in Python? Provide an example.
21. Explain what a lambda function is in Python.
22. What is the purpose of the 'pass' statement in Python?
23. Describe the distinction between a shallow copy and a deep copy in Python.
24. Which sorting technique is employed by Python's 'sort()' and 'sorted()' functions?
25. What are Decorators in Python?
26. Define Iterators in Python.
27. What are Generators in Python?
28. Does Python support multiple inheritance?
29. Explain Polymorphism in Python.
30. Define encapsulation in Python.

31. How do you achieve data abstraction in Python?
32. Describe the memory management approach used in Python.
33. How can you delete a file using Python?
34. What is slicing in Python?
35. What is Dictionary Comprehension in Python? Can you provide an example?
36. Is there Tuple Comprehension in Python? If so, how does it work, and if not, why?
37. What distinguishes a List from a Tuple?
38. Explain the difference between a shallow copy and a deep copy in Python.
39. What is Python's Switch Statement, and how does it work?
40. What is the purpose of the Walrus Operator in Python?
41. What are the Built-in data types in Python?
42. What does the 'swapcase()' function do in Python?
43. Explain the significance of 'init()' in Python.
44. Can you write code to display the current time in Python?
45. What is PIP in Python, and what is its role?
46. Describe the purpose of the 'zip' function in Python.
47. What are Pickling and Unpickling in Python?
48. Define monkey patching in Python.
49. Explain the concept of Function Annotations in Python.
50. What are Exception Groups in Python?
51. What is the significance of PYTHONPATH in Python?
52. What are unit tests in Python?
53. What is the Global Interpreter Lock (GIL) in Python?
54. How do you debug a Python program?
55. Explain the concepts of Class and Object in Python.
56. How does inheritance work in Python, and can you provide an example?
57. Are access specifiers used in Python?
58. Differentiate between the 'new' and 'override' modifiers.
59. Why is the 'finalize' method used in Python?
60. What is the 'main' function in Python, and how do you invoke it?
61. Describe how to create a class in Python.
62. What do you know about pandas in Python?
63. Define a pandas data frame.
64. How can you combine different pandas' data frames?
65. Can you create a series from a dictionary object in pandas?
66. How do you identify and handle missing values in a data frame?
67. Explain the concept of reindexing in pandas.
68. How can you add a new column to a panda's data frame?
69. What methods are available to delete indices, rows, and columns from a data frame in pandas?
70. How can you get items from series A that are not present in series B?

71. How can you obtain items that are unique to both series A and B?
72. Does the panda's library recognize dates when importing data from different sources?
73. Explain the concept of NumPy in Python.
74. What advantages do NumPy arrays offer over Python lists?
75. What steps are involved in creating 1D, 2D, and 3D arrays in NumPy?
76. Given a numpy array and a new column, how do you delete the second column and replace it with a new column value?
77. How do you efficiently load data from a text file in NumPy?
78. How do you read CSV data into an array in NumPy?
79. How do you sort an array based on the Nth column in NumPy?
80. How do you find the nearest value in each numpy array?
81. How do you reverse a numpy array using a single line of code?
82. How do you determine the shape of a given NumPy array?
83. Differentiate between a package and a module in Python.
84. List some commonly used built-in modules in Python.
85. What are lambda functions, and how are they used in Python?
86. How can you generate random numbers in Python?
87. Can you check if all characters in a string are alphanumeric in Python?
88. Define GIL in Python.
89. Explain the significance of PYTHONPATH in Python.
90. What is PIP, and how is it used in Python?
91. Are there any tools available for identifying bugs and performing static analysis in Python?
92. What distinguishes deep copies from shallow copies in Python?
93. How is Python code organized into files and directories?
94. Why should one choose Python as a programming language?
95. What types of applications can Python be used for?
96. What are the key advantages of using Python?
97. Define Python literals.
98. Describe Python Functions.
99. What is the purpose of the 'zip()' capability in Python?
100. How does Python's parameter passing system work?
101. How do you overload methods or constructors in Python?
102. What is the difference between the 'remove()' function and the 'del' statement in Python?
103. How do you remove whitespaces from a string in Python?
104. How do you remove leading whitespaces from a string in Python?
105. Why is the 'join()' function used in Python?
106. Provide an example of the 'shuffle()' method in Python.
107. Explain the use of the 'break' statement in Python.
108. Define a tuple in Python.
109. List some file-related libraries/modules in Python.

110. What are the different file processing modes supported by Python?
111. What are operators in Python, and how are they categorized?
112. How can you create a Unicode string in Python?
113. Is Python an interpreted language?
114. What are the rules governing local and global variables in Python?
115. Describe the concept of a namespace in Python.
116. What are iterators in Python?
117. What is a generator in Python?
118. Explain the concept of 'Pass' in Python.
119. Describe docstrings in Python.
120. What are negative indices in Python, and why are they used?
121. What is pickling and unpickling in Python?
122. When choosing between Java and Python, what factors should be considered?
123. What is the usage of the 'help()' and 'dir()' functions in Python?
124. How does Python perform Compile-time and Run-time code checking?
125. What is the shortest method for opening a text file and displaying its content in Python?
126. What is the purpose of the 'enumerate()' function in Python?
127. Given the list A=[10,40,16,17,29,66,2,94], what would be the output of A[3]?
128. What is type conversion in Python?
129. How do you send an email in Python Language?
130. Can you create a program to add two integers greater than 0 without using the plus operator?
131. Create a program to convert dates from yyyy-mm-dd to dd-mm-yyyyy.
132. Create a program that combines two dictionaries, summing values for similar keys to create a new dictionary.
133. Is there a built-in do-while loop in Python?
134. What types of joins are offered by Pandas for data manipulation?
135. How are data frames merged in Pandas?
136. What is the best way to obtain the first five entries of a Pandas data frame?
137. How can you access the latest five entries in a Pandas data frame?
138. Explain the concept of a classifier in Python.

67

Sample Python Programs

1. **Write a Python program to calculate the sum, product, division, multiplication, and modular division of two numbers entered by the user.**

```
# Input the two numbers from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Calculate the sum of the two numbers
sum_result = num1 + num2

# Calculate the product of the two numbers
product_result = num1 * num2

# Calculate the division of the two numbers
division_result = num1 / num2

# Calculate the multiplication of the two numbers
multiplication_result = num1 * num2

# Calculate the modulus (remainder) of the two numbers
modulus_result = num1 % num2

# Print the results
print("Sum:", sum_result)
print("Product:", product_result)
print("Division:", division_result)
print("Multiplication:", multiplication_result)
print("Modulus:", modulus_result)
```

2. **Write a Python program to check if a given number is even or odd.**

```
number = int(input("Enter a number: "))
if  number % 2 == 0:
   print(number, "is even.")
```

```
else:
    print(number, "is odd.")
```

3. **Write a Python program to convert temperature in Celsius to Fahrenheit.**

```
celsius = float(input("Enter temperature in Celsius: "))
fahrenheit = (celsius * 9/5) + 32
print("Temperature in Fahrenheit:", fahrenheit)
```

4. **Write a Python program to calculate the factorial of a given number.**

```
number = int(input("Enter a number: "))
factorial = 1
if number < 0:
   print("Factorial cannot be calculated for negative numbers.")
elif number == 0:
   print("The factorial of 0 is 1.")
else:
   for i in range(1, number + 1):
      factorial *= i
print("The factorial of", number, "is", factorial)
```

5. **Write a Python program to find the area of a triangle given its base and height.**

```
base = float(input("Enter the base of the triangle: "))
height = float(input("Enter the height of the triangle: "))
area = (base * height) / 2
print("The area of the triangle is:", area)
```

6. **Write a Python Program to find the simple interest.**

```
# Input the principal amount, interest rate, and time period from the user
principal = float(input("Enter the principal amount: "))
rate = float(input("Enter the interest rate: "))
time = float(input("Enter the time period (in years): "))

# Calculate the simple interest
```

```
interest = (principal * rate * time) / 100

# Print the result
print("The simple interest is:", interest)
```

7. Write a python program to swap two numbers using all 3 approaches.

Approach 1 : Using a temporary variable

```
# Input the two numbers from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Swap the numbers using a temporary variable
temp = num1
num1 = num2
num2 = temp

# Print the swapped numbers
print("After swapping:")
print("First number:", num1)
print("Second number:", num2)
```

Approach 2: Using arithmetic operations.

```
# Input the two numbers from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Swap the numbers using arithmetic operations
num1 = num1 + num2
num2 = num1 - num2
num1 = num1 - num2

# Print the swapped numbers
print("After swapping:")
print("First number:", num1)
print("Second number:", num2)
```

Approach 3: Using multiple assignment

```
# Input the two numbers from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Swap the numbers using multiple assignment
num1, num2 = num2, num1

# Print the swapped numbers
print("After swapping:")
print("First number:", num1)
print("Second number:", num2)
```

8. Write a program to find the sum of n natural numbers.

Approach 1:

```
# Input the value of n from the user
n = int(input("Enter a positive integer: "))

# Calculate the sum of the first n natural numbers
sum_natural = (n * (n + 1)) // 2

# Print the result
print("The sum of the first", n, "natural numbers is:", sum_natural)
```

Approach 2:

```
# Input the value of n from the user
n = int(input("Enter a positive integer: "))

# Initialize variables
sum_natural_numbers = 0
count = 1

# Calculate the sum of the first n natural numbers using a while loop
while count <= n:
    sum_natural_numbers += count
    count += 1
```

```
# Print the result
print("The sum of the first", n, "natural numbers is:", sum_natural_numbers)
```

Approach 3:

```
# Input the value of n from the user
n = int(input("Enter a positive integer: "))

# Initialize variable
sum_natural_numbers = 0

# Calculate the sum of the first n natural numbers using a for loop
for i in range(1, n + 1):
    sum_natural_numbers += i

# Print the result
print("The sum of the first", n, "natural numbers is:",
sum_natural_numbers)
```

9. Python program to calculate the area of rectangle and triangle print the result.

```
# Calculate the area of a rectangle and a triangle

# Rectangle
width = float(input("Enter the width of the rectangle: "))
height = float(input("Enter the height of the rectangle: "))
rectangle_area = width * height
print("The area of the rectangle is:", rectangle_area)

# Triangle
base = float(input("Enter the base length of the triangle: "))
height_triangle = float(input("Enter the height of the triangle: "))
triangle_area = 0.5 * base * height_triangle
print("The area of the triangle is:", triangle_area)
```

#Ouput:

```
Enter the width of the rectangle: 10
Enter the height of the rectangle: 32
The area of the rectangle is: 320.0
```

Enter the base length of the triangle: 3
Enter the height of the triangle: 4
The area of the triangle is: 6.0

10. Write a python program to find the sum of all numbers, odd numbers and even numbers upto n.

```
# Input
n = int(input("Enter a number: "))

# Sum of all numbers, odd numbers, and even numbers
sum_all = 0
sum_odd = 0
sum_even = 0

for num in range(1, n + 1):
    sum_all += num
    if num % 2 == 1:
        sum_odd += num
    else:
        sum_even += num
print(f"Sum of all numbers up to {n}: {sum_all}")
print(f"Sum of odd numbers up to {n}: {sum_odd}")
print(f"Sum of even numbers up to {n}: {sum_even}")
```

#Output:

Enter a number: 10
Sum of all numbers up to 10: 55
Sum of odd numbers up to 10: 25
Sum of even numbers up to 10: 30

11. Python Program to find the largest of 3 numbers

```
# Input
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
num3 = float(input("Enter the third number: "))

# Finding the largest number
if num1 >= num2:
    if num1 >= num3:
        largest = num1
```

```
    else:
        largest = num3
else:
    if num2 >= num3:
        largest = num2
    else:
        largest = num3

print(f"The largest number among {num1}, {num2}, and {num3} is:
{largest}")
```

#Output

```
Enter the first number: 12
Enter the second number: -3
Enter the third number: 34
The largest number among 12.0, -3.0, and 34.0 is: 34.0
```

12. Python program to perform basic calculations

```
# Sample numbers
num1 = int(input("Enter first number"))
num2 = int(input("Enter second number"))

# Calculate the sum of two numbers
sum_result = num1 + num2
print("Sum:", sum_result)

# Calculate the product of two numbers
product_result = num1 * num2
print("Product:", product_result)

# Calculate the difference of two numbers
difference_result = num1 - num2
print("Difference:", difference_result)

# Calculate the division of two numbers
division_result = num1 / num2
print("Division:", division_result)

# Calculate the integer division of two numbers
integer_division_result = num1 // num2
```

```
print("Integer Division:", integer_division_result)

# Calculate the modulo division of two numbers
modulo_division_result = num1 % num2
print("Modulo Division:", modulo_division_result)

# Output
Enter first number 3
Enter second number 2
Sum: 5
Product: 6
Difference: 1
Division: 1.5
Integer Division: 1
Modulo Division: 1
```

13. Python program to check whether the given number is prime or not.

```
# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

# Input
num = int(input("Enter a number: "))

# Check and print result
if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")

#Output
Enter a number: 7
7 is a prime number.
```

14. Python Program to print the pascal triangle

```
# Function to print Pascal's triangle
def print_pascals_triangle(n):
    for line in range(1, n + 1):
        num = 1
        for i in range(1, n - line + 1):
            print(" ", end="")

        for j in range(1, line + 1):
            print(num, end=" ")
            num = num * (line - j) // j

        print()

        # Input
rows = int(input("Enter the number of rows: "))

# Print Pascal's triangle
print_pascals_triangle(rows)
```

#Output:

```
Enter the number of rows: 5
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

15. Python Program to generate Fibonacci series upto n.

```
# Function to generate Fibonacci series up to n
def generate_fibonacci(n):
    a, b = 0, 1
    while a <= n:
        print(a, end=" ")
        a, b = b, a + b

# Input
limit = int(input("Enter the limit for Fibonacci series: "))
```

```
# Generate and print Fibonacci series
print("Fibonacci series up to", limit, ": ", end="")
generate_fibonacci(limit)

#ouput:
Enter the limit for Fibonacci series: 10
Fibonacci series up to 10 : 0 1 1 2 3 5 8
```

16. Implement a code which prompts the user for Celsius temperature, convert the temperature to Fahrenheit and print out the converted temperature by handling the exception.

```
# Function to convert Celsius to Fahrenheit
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

try:
    celsius_temp = float(input("Enter the temperature in Celsius: "))
    fahrenheit_temp = celsius_to_fahrenheit(celsius_temp)
    print(f"The temperature {celsius_temp}°C is equivalent to
{fahrenheit_temp:.2f}°F.")
except ValueError:
    print("Invalid input. Please enter a valid numeric value for
temperature.")
except Exception as e:
    print("An error occurred:", e)

#Output
Enter the temperature in Celsius: 100
The temperature 100.0°C is equivalent to 212.00°F.
```

17. Explain Negative Integer Modulo division in python

In Python, the modulo operator (%) calculates the remainder of the division between two integers. When dealing with negative numbers, the behavior of the modulo operation might not be immediately intuitive, but it follows a consistent rule.

Positive Dividend with Negative Divisor

```
result = 7 % -3
print(result)  # Output: -2
```

In this case, 7 % -3 calculates the remainder when 7 is divided by -3. The result is -2 because -2 is the remainder that, when added to -3, gives you 7. Mathematically, -3 * -3 + (-2) equals 7.

Negative Dividend with Positive Divisor

```
result = -7 % 3
print(result)  # Output: 2
```

Here, -7 % 3 calculates the remainder when -7 is divided by 3. The result is 2 because 2 is the remainder that, when added to 3, gives you -7. Mathematically, 3 * -3 + 2 equals -7.

Negative Dividend with Negative Divisor:

```
result = -7 % -3
print(result)  # Output: -1
```

In this case, -7 % -3 calculates the remainder when -7 is divided by -3. The result is -1 because -1 is the remainder that, when added to -3, gives you -7. Mathematically, -3 * 3 + (-1) equals -7.

The behavior of the modulo operation ensures that the quotient and remainder, when recombined, reconstruct the original dividend, regardless of the sign of the numbers involved. Keep in mind that the sign of the remainder might not always match the sign of the dividend or divisor, which can sometimes be counter intuitive

68

Top Python programming Questions

1. Program to check whether the given number is odd or even.
2. Program to check whether the given string is Palindrome or not.
3. Program to compute the Factorial of a given number.
4. Program to generate Fibonacci Series up to n.
5. Program to determine whether the given number is Armstrong Number or not.
6. Program to simulate simple Calculator.
7. Program to check whether the given year or not.
8. Program to check whether the given number is Prime or not.
9. Program to find Area of triangle In Python
10. Program to Reverse a List
11. Program to Reverse a Number
12. Program to Swap two numbers.
13. Program to Reverse a String
14. Program to Print the Fibonacci Series
15. Program to check if the given strings are anagram or not.
16. Program to find a maximum of two numbers.
17. Program to find a minimum of two numbers.
18. Program to find GCD of two numbers.
19. Program to Solve Quadratic Equation
20. Program to Generate a Random Number
21. Program to Find the Largest Among Three Numbers
22. Program to Find Numbers Divisible by Another Number
23. Program to Count the Number of Each Vowel
24. Program to Merge Mails
25. Program to Merge Two Dictionaries
26. Program to Safely Create a Nested Directory
27. Program to Get Line Count of a File
28. Program to Count the Number of Digits Present in a Number.
29. Program to Remove Duplicate Element from a List.
30. Program to do arithmetical operations.
31. Program to solve quadratic equation.
32. Program to swap two variables.

33. Program to generate a random number.
34. Program to convert kilometers to miles.
35. Program to convert Celsius to Fahrenheit.
36. Program to display calendar.
37. Program to Check if a Number is Positive, Negative or Zero.
38. Program to Print all Prime Numbers in an Interval.
39. Program to Display the multiplication Table.
40. Program to Print the Fibonacci sequence.
41. Program to Check Armstrong Number.
42. Program to Find Armstrong Number in an Interval
43. Program to Find the Sum of Natural Numbers.
44. Program to Convert Decimal to Binary, Octal and Hexadecimal
45. Program to Find ASCII value of a character.
46. Program to Make a Simple Calculator.
47. Program to Display Calendar.
48. Program to Display Fibonacci Sequence Using Recursion.
49. Program to Find Factorial of Number Using Recursion.
50. Program to check if the given number is a Disarium Number.
51. Program to check if the given number is Happy Number.
52. Program to copy all elements of one array into another array.
53. Program to find the frequency of each element in the array.
54. Program to left rotate the elements of an array.
55. Program to print the duplicate elements of an array.
56. Program to print the elements of an array.
57. Program to print the elements of an array in reverse order.
58. Program to print the elements of an array present on even position.
59. Program to print the elements of an array present on odd position.
60. Program to print the largest element in an array.
61. Program to print the smallest element in an array.
62. Program to print the number of elements present in an array.
63. Program to print the sum of all elements in an array.
64. Program to right rotate the elements of an array.
65. Program to sort the elements of an array in ascending order.
66. Program to sort the elements of an array in descending order.
67. Program to Add Two Matrices
68. Program to Multiply Two Matrices
69. Program to Transpose a Matrix
70. Program to Sort Words in Alphabetic Order
71. Program to Remove Punctuation From a String
72. Program to reverse a string
73. Program to convert list to string
74. Program to convert int to string
75. Program to concatenate two strings

76. Program to generate a Random String
77. Program to convert Bytes to string
78. Program to append element in the list
79. Program to compare two lists
80. Program to convert list to dictionary
81. Program to remove an element from a list
82. Program to add two lists
83. Program to convert List to Set
84. Program to convert list to string
85. Program to create a dictionary
86. Program to convert list to dictionary
87. Program to sort a dictionary
88. Program to Merge two Dictionaries
89. Binary Search in Python
90. Linear Search in Python
91. Bubble Sort in Python
92. Insertion Sort in Python
93. Heap Sort in Python
94. Merge Sort in Python
95. Program to Check Whether a Given Number is Even or Odd using Recursion
96. Program to Print All Odd Numbers in a Range
97. Program to Check if a Number is a Palindrome
98. Program to Reverse a Number
99. Program to Print All Integers that Aren't Divisible by Either 2 or 3
100. Program to Find Numbers which are Divisible by 7 and Multiple of 5 in a Given Range
101. Program to Print All Numbers in a Range Divisible by a Given Number
102. Program to Find Sum of Digits of a Number
103. Program to Find Sum of Digit of a Number using Recursion
104. Program to Find Sum of Digit of a Number Without Recursion
105. Program to Count the Number of Digits in a Number
106. Program to Find All the Divisors of an Integer
107. Program to Find the Smallest Divisor of an Integer
108. Program to Print Binary Equivalent of an Integer using Recursion
109. Program to Print Binary Equivalent of a Number without Using Recursion
110. Program to Print Table of a Given Number
111. Program to Calculate Grade of a Student
112. Program to Check if a Date is Valid and Print the Incremented Date if it is
113. Program to Check Whether a given Year is a Leap Year
114. Program to Convert Centimeters to Feet and Inches
115. Program to Read a Number n and Compute n+nn+nnn
116. Program to Check Whether a Given Number is Perfect Number

117. Program to Check if a Number is a Strong Number
118. Program to Print Numbers in a Range Without using Loops.
119. Program to Find the Prime Factors of a Number
120. Program to Check If Two Numbers are Amicable Numbers or Not
121. Program to Find Whether a Number is a Power of Two
122. Program to Calculate the Power using Recursion
123. Program to Find Product of Two Numbers using Recursion
124. Program to Find All Perfect Squares in the Given Range
125. Program to Print All Possible Combinations of Three Digits
126. Factorial & Fibonacci Programs in Python
127. Program to Find Fibonacci Numbers using Recursion
128. Program to Find the Fibonacci Series Without using Recursion
129. Program to Find the Factorial of a Number using Recursion
130. Program to Convert Binary to Gray Code
131. Program to Print an Inverted Star Pattern
132. Program to Print Pascal Triangle
133. Program to Find the Roots of a Quadratic Equation
134. Program to Find Quotient and Remainder of Two Numbers
135. Program to Find All Pythagorean Triplets in the Range
136. Program to Compute a Polynomial Equation
137. Program to Swap Two Numbers without using Third Variable
138. Program to Find Sum of Series 1 + 1/2 + 1/3 + 1/4 + ……. + 1/N
139. Program to Find the Sum of the Series 1/1!+1/2!+1/3!+…1/N!
140. Program to Find the Sum of the Series: 1 + x^2/2 + x^3/3 + … x^n/n
141. Program to Read a Number n and Print the Series “1+2+…..+n= “
142. Program to Find the Sum of Sine Series
143. Program to Find the Sum of Cosine Series
144. Program to Find the GCD and LCM of Two Numbers
145. Python Program to Find the GCD and LCM of Two Numbers using Recursion.
146. Program to Check if a String is a Pangram or Not
147. Program to Remove Odd Indexed Characters in a string.
148. Program to Reverse a String using Recursion.
149. Program to Reverse a String Without using Recursion.
150. Program to Find the Length of a String without Library Function.
151. Program to Count the Number of Words and Characters in a String.
152. Program to Count Number of Lowercase Characters in a String.
153. Program to Count the Number of Vowels in a String.
154. Program to Count Number of Uppercase and Lowercase Letters in a String.
155. Program to Count the Number of Digits and Letters in a String.
156. Program to Check if the Substring is Present in the Given String
157. Program to Find Common Characters in Two Strings

158. Program to Print All Letters Present in Both Strings
159. Program that Displays which Letters are in First String but not in Second.
160. Program that Displays Letters that are not Common in Two Strings.
161. Program to Create a New String Made up of First and Last 2 Characters.
162. Program to Find the Larger String without using Built-in Functions.
163. Program to Swap the First and the Last Character of a String.
164. Program to Sort Hyphen Separated Sequence of Words in Alphabetical Order.
165. Program to Count the Occurrences of Each Word in a String.
166. Program to Count Number of Vowels in a String using Sets.
167. Program to Check if a Given String is Palindrome.
168. Program to Check whether two Strings are Anagrams.
169. Program to Check whether a String is Palindrome or not using Recursion.
170. Program to Find All Odd Palindrome Numbers in a Range without using Recursion.
171. Program to Create Dictionary from an Object.
172. Program to Check if a Key Exists in a Dictionary or Not.
173. Program to Add a Key-Value Pair to the Dictionary.
174. Program to Find the Sum of All the Items in a Dictionary
175. Program to Multiply All the Items in a Dictionary
176. Program to Remove a Key from a Dictionary
177. Program to Concatenate Two Dictionaries
178. Program to Map Two Lists into a Dictionary
179. Program to Create a Dictionary with Key as First Character and Value as Words Starting with that Character
180. Program to Create Dictionary that Contains Number
181. Program to Count the Frequency of Each Word in a String using Dictionary.
182. Program to Create a List of Tuples with the First Element as the Number and Second Element as the Square of the Number
183. Program to Remove All Tuples in a List Outside the Given Range
184. Program to Sort a List of Tuples in Increasing Order by the Last Element in Each Tuple
185. Python Program to Create a Class which Performs Basic Calculator Operations
186. Python Program to Append, Delete and Display Elements of a List using Classes
187. Python Program to Find the Area of a Rectangle using Classes
188. Python Program to Find the Area and Perimeter of the Circle using Class
189. Python Program to Create a Class in which One Method Accepts a String from the User and Another Prints it
190. Python Program to Create a Class and Get All Possible Distinct Subsets from a Set

191. Python Programs on File Handling
192. Python Program to Read the Contents of the File
193. Python Program to Copy One File to Another File
194. Python Program to Count the Number of Lines in Text File
195. Python Program to Count the Number of Blank Spaces in a Text File
196. Python Program to Count the Occurrences of a Word in a Text File
197. Python Program to Count the Number of Words in a Text File
198. Python Program to Capitalize First Letter of Each Word in a File
199. Python Program to Counts the Number of Times a Letter Appears in the Text File
200. Python Program to Extract Numbers from Text File
201. Python Program to Print the Contents of File in Reverse Order
202. Python Program to Append the Content of One File to the End of Another File
203. Python Program to Read a String from the User and Append it into a File
204. Python Programming Examples on List
205. Python Program to Find Largest Number in a List
206. Python Program to Find Second Largest Number in a List
207. Python Program to Print Largest Even and Largest Odd Number in a List
208. Python Program to Split Even and Odd Elements into Two Lists
209. Python Program to Find Average of a List
210. Python Program to Print Sum of Negative Numbers, Positive Even & Odd Numbers in a List
211. Python Program to Count Occurrences of Element in List
212. Python Program to Find the Sum of Elements in a List using Recursion
213. Python Program to Find the Length of a List using Recursion
214. Python Program to Merge Two Lists and Sort it
215. Python Program to Remove Duplicates from a List
216. Python Program to Swap the First and Last Element in a List
217. Python Program to Sort a List According to the Second Element in Sublist
218. Python Program to Return the Length of the Longest Word from the List of Words
219. Python Program to Find the Number Occurring Odd Number of Times in a List
220. Python Program to Generate Random Numbers from 1 to 20 and Append Them to the List
221. Python Program to Remove the ith Occurrence of the Given Word in a List
222. Python Program to Find the Cumulative Sum of a List
223. Python Program to Find the Union of Two Lists
224. Python Program to Find the Intersection of Two Lists
225. Python Program to Flatten a List without using Recursion
226. Python Program to Find the Total Sum of a Nested List Using Recursion
227. Python Program to Flatten a Nested List using Recursion

228. Python Programs on File Handling
229. Python Program to Read the Contents of the File
230. Python Program to Copy One File to Another File
231. Python Program to Count the Number of Lines in Text File
232. Python Program to Count the Number of Blank Spaces in a Text File
233. Python Program to Count the Occurrences of a Word in a Text File
234. Python Program to Count the Number of Words in a Text File
235. Python Program to Capitalize First Letter of Each Word in a File
236. Python Program to Counts the Number of Times a Letter Appears in the Text File
237. Python Program to Extract Numbers from Text File
238. Python Program to Print the Contents of File in Reverse Order
239. Python Program to Append the Content of One File to the End of Another File
240. Python Program to Read a String from the User and Append it into a File

69

Sample Projects

1. Guess the Number

```
import random
number = random.randint(1, 10)

player_name = input("Hello, what is your name? ")
number_of_guesses = 0
print('I\'m glad to meet you! {} \nLet\'s play a game with you, I will think a
number between 1 and 10 then you will guess, alright? \nDon\'t forget! You
have only 3 chances so guess:'.format(player_name))

while number_of_guesses < 3:
   guess = int(input())
   number_of_guesses += 1
   if guess < number:
      print('Your estimate is too low, go up a little!')
   if guess > number:
      print('Your estimate is too high, go down a bit!')
   if guess == number:
      break
if guess == number:
   print( 'Congratulations {}, you guessed the number in {}
tries!'.format(player_name, number_of_guesses))
else:
   print('Close but no chocklet, you couldn\'t guess the number. \nWell, the
number was {}.'.format(number))
```

Sample Output

Hello, what is your name? Thyagu
I'm glad to meet you! Thyagu
Let's play a game with you, I will think a number between 1 and 10 then you will guess, alright?
Don't forget! You have only 3 chances so guess:
10
Your estimate is too high, go down a bit!

```
5
Your estimate is too low, go up a little!
7

Your estimate is too high, go down a bit!
Close but no chocklet, you couldn't guess the number.
Well, the number was 6.
```

2. Magic 8 Ball with a List

The Magic 8 Ball problem refers to a hypothetical situation in computer science where a program needs to make a decision based on incomplete or ambiguous information. It is named after the Magic 8 Ball toy, which is a popular fortune-telling device that provides random answers to yes-or-no questions. The following program illustrates the Magic 8 Ball for yes or no questions.

```
import random
messages = ['It is certain',
 'It is decidedly so',
 'Yes definitely',
 'Reply hazy try again',
 'Ask again later',
 'Concentrate and ask again',
 'My reply is no',
 'Outlook not so good',
 'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

#Output: Any one of the messages like, ‘Concentrate and ask again’

3. Password Locker

A simple password locker in Python is a basic program that allows users to store and retrieve passwords for different accounts. In this explanation, I'll walk you through the fundamental components and functionality of a simple password locker.

Data Storage

The password locker needs a way to store the account names and their corresponding passwords. In this example, we'll use a Python dictionary to store

this data, where the account names will act as keys, and the passwords will be the values.

```
passwords =
{
        'email': 'password1',
        'social_media': 'password2',
         'bank': 'password3'
}
```

Functionality

The password locker should provide several functionalities, such as:

1. Saving a new password for an account.
2. Retrieving a password for a specific account.
3. Listing all the accounts and their associated passwords.
4. Quitting the program.

Save Password Function

We need a function to save a new password to the password locker. This function takes the account name and password as input and stores it in the passwords dictionary.

```
        def save_password(account, password):
           passwords[account] = password
```

Retrieve Password Function

To retrieve a password, we need a function that takes the account name as input, looks it up in the passwords dictionary, and returns the password if it exists. We can use the pyperclip library to copy the password to the clipboard for easy access.

```
import pyperclip

        def retrieve_password(account):
           if account in passwords:
              pyperclip.copy(passwords[account])
              print(f"Password for '{account}' copied to clipboard.")
           else:
              print(f"No password found for '{account}'.")
```

List Accounts Function

We also want to list all the accounts stored in the password locker.

```
def list_accounts():
    print("Stored Accounts:")
    for account in passwords:
        print(account)
```

Main Program Loop

We'll create a simple loop to present a menu to the user and allow them to interact with the password locker. The user can choose the actions they want to perform, such as saving, retrieving, listing, or quitting.

```
while True:
    print("\nPassword Locker")
    print("1. Save a password")
    print("2. Retrieve a password")
    print("3. List all accounts")
    print("4. Quit")

    choice = input("\nEnter your choice (1-4): ")

    if choice == '1':
        account = input("Enter the account name: ")
        password = input("Enter the password: ")
        save_password(account, password)
        print(f"Password for '{account}' saved.")

    elif choice == '2':
        account = input("Enter the account name: ")
        retrieve_password(account)

    elif choice == '3':
        list_accounts()

    elif choice == '4':
        break

    else:
        print("Invalid choice. Please try again.")
```

```
print("Password Locker closed.")
```

Installing pyperclip

Install the pyperclip library to use the clipboard functionality. You can install it using pip as follows:

pip install pyperclip

Complete Source Code

```
import pyperclip

passwords =
{
        'email': 'password1',
        'social_media': 'password2',
         'bank': 'password3'
}

def save_password(account, password):
   passwords[account] = password

def retrieve_password(account):
   if account in passwords:
      pyperclip.copy(passwords[account])
      print(f"Password for '{account}' copied to clipboard.")
   else:
      print(f"No password found for '{account}'.")

def list_accounts():
   print("Stored Accounts:")
   for account in passwords:
      print(account)

# Main program loop
while True:
   print("\nPassword Locker")
   print("1. Save a password")
   print("2. Retrieve a password")
   print("3. List all accounts")
```

```
        print("4. Quit")

        choice = input("\nEnter your choice (1-4): ")

        if choice == '1':
            account = input("Enter the account name: ")
            password = input("Enter the password: ")
            save_password(account, password)
            print(f"Password for '{account}' saved.")

        elif choice == '2':
            account = input("Enter the account name: ")
            retrieve_password(account)

        elif choice == '3':
            list_accounts()
        elif choice == '4':
            break
        else:
            print("Invalid choice. Please try again.")
    print("Password Locker closed.")
```

Sample Output

Password Locker
1. Save a password
2. Retrieve a password
3. List all accounts
4. Quit

Enter your choice (1-4): 3
Stored Accounts:
email
social_media
bank

Password Locker
1. Save a password
2. Retrieve a password
3. List all accounts
4. Quit

Enter your choice (1-4): 1

Enter the account name: email
Enter the password: 123456
Password for 'email' saved.

Password Locker
1. Save a password
2. Retrieve a password
3. List all accounts
4. Quit

Enter your choice (1-4): 4
Password Locker closed.

4. Adding Bullets to Wiki Markup

The project "Adding Bullets to Wiki Markup in Python" involves writing a Python program that adds bullet points to lines of text in the Wiki markup format. The program should identify lines that need bullet points and modify them accordingly.

Here's an explanation of the steps involved in this project and an example implementation:

Understanding Wiki Markup and Bullet Points

Wiki markup is a lightweight markup language used on wikis to format and structure text. Bullet points in Wiki markup are typically represented using an asterisk (*) character at the beginning of a line.

Input and Output

The program should take input in the form of plain text and output the modified text with added bullet points.

Algorithm and Implementation

Here's a step-by-step algorithm to implement the program:
1. Read the input text.
2. Split the text into individual lines.
3. Iterate over each line of the text.
4. Check if a line does not start with an asterisk (*) character.
5. If the line does not start with an asterisk, add the asterisk at the beginning of the line.

6. Join the modified lines back together into a single string.
7. Output the modified text.

Here's an example implementation of the project:

```
def add_bullet_points(text):
    lines = text.split('\n')
    modified_lines = []

    for line in lines:
        if not line.startswith('*'):
            modified_lines.append('* ' + line)
        else:
            modified_lines.append(line)

    modified_text = '\n'.join(modified_lines)
    return modified_text

# Example usage
input_text = """
This is a paragraph of text.
This is another line of text.
This line needs a bullet point.
Another line without a bullet point.
"""

modified_text = add_bullet_points(input_text)
print(modified_text)
```

In this example, the **add_bullet_points()** function takes the input text, splits it into individual lines using the newline character ('\n'), and iterates over each line. If a line does not start with an asterisk, it adds the asterisk followed by a space ('* ') at the beginning of the line. Finally, it joins the modified lines back together using the newline character and returns the modified text.

The example usage demonstrates how to call the add_bullet_points() function with the input text and print the modified text.

You can customize this implementation according to your specific needs, such as reading input from a file, applying more complex rules for bullet points, or incorporating additional formatting features of Wiki markup.

Sample Output

```
* This is a paragraph of text.
* This is another line of text.
* This line needs a bullet point.
* Another line without a bullet point.
*
```

5. Generating Random Quiz Files

```
import os
import random

# List of quiz questions and their answers
quiz_data = [
{
        "question": "What is the capital of France?",
        "answer": "Paris"
},
{
        "question": "Which planet is known as the 'Red Planet'?",
        "answer": "Mars"
},
{
        "question": "What is the largest mammal on Earth?",
         "answer": "Blue Whale"
}
]

def generate_quiz_file(quiz_number):
   quiz_name = f"quiz_{quiz_number}.txt"

   with open(quiz_name, "w") as file:
      file.write(f"Quiz {quiz_number}\n\n")
      for i, qna in enumerate(quiz_data):
         question = qna["question"]
         answer = qna["answer"]
         file.write(f"Question {i + 1}: {question}\n")
         file.write(f"Answer: \n\n")

   print(f"Generated quiz file: {quiz_name}")
```

```
if __name__ == "__main__":
    for quiz_number in range(1, 4):  # Generate 3 quiz files
        generate_quiz_file(quiz_number)
```

Sample Output

Generated quiz file: quiz_1.txt
Generated quiz file: quiz_2.txt
Generated quiz file: quiz_3.txt

In this program, we define a list called quiz_data containing dictionaries for each quiz question and answer. The generate_quiz_file function creates a quiz file for each quiz number, writing the questions and leaving space for answers. The program generates three quiz files named quiz_1.txt, quiz_2.txt, and quiz_3.txt.

6. Multiclipboard

```
import shelve
import pyperclip
import sys

def copy_to_multiclipboard(shelf, key, text):
    shelf[key] = text
    print(f"Text copied to clipboard '{key}': {text}")

def paste_from_multiclipboard(shelf, key):
    if key in shelf:
        text = shelf[key]
        pyperclip.copy(text)
        print(f"Text from clipboard '{key}' pasted: {text}")
    else:
        print(f"Clipboard '{key}' is empty")

if __name__ == "__main__":
    with shelve.open("multiclipboard") as shelf:
        while True:
            print("\nOptions:")
            print("1. Copy to clipboard")
            print("2. Paste from clipboard")
            print("3. Exit")
```

```
    choice = input("Enter your choice (1/2/3): ")
    if choice == "1":
        key = input("Enter a key for the clipboard entry: ")
        text = input("Enter the text to copy: ")
        copy_to_multiclipboard(shelf, key, text)
    elif choice == "2":
        key = input("Enter the key of the clipboard entry: ")
        paste_from_multiclipboard(shelf, key)
    elif choice == "3":
        print("Exiting the Multiclipboard program.")
        sys.exit()
    else:
        print("Invalid choice. Please select a valid option.")
```

Sample Output
Options
1. Copy to clipboard
2. Paste from clipboard
3. Exit

Enter your choice (1/2/3): 1
Enter a key for the clipboard entry: key1
Enter the text to copy: Sample text 1
Text copied to clipboard 'key1': Sample text 1

Options
1. Copy to clipboard
2. Paste from clipboard
3. Exit

Enter your choice (1/2/3): 1
Enter a key for the clipboard entry: key2
Enter the text to copy: Sample text 2
Text copied to clipboard 'key2': Sample text 2

Options
1. Copy to clipboard
2. Paste from clipboard
3. Exit

Enter your choice (1/2/3): 2
Enter the key of the clipboard entry: key1

Text from clipboard 'key1' pasted: Sample text 1

Options
1. Copy to clipboard
2. Paste from clipboard
3. Exit

Enter your choice (1/2/3): 2
Enter the key of the clipboard entry: key3
Clipboard 'key3' is empty

Options
1. Copy to clipboard
2. Paste from clipboard
3. Exit

Enter your choice (1/2/3): 3
Exiting the Multiclipboard program.

7. Renaming Files with American-Style Dates to European-Style Dates

```
import os
import re
from datetime import datetime

def rename_files_with_dates(folder_path):
    # Get a list of all files in the folder
    file_list = os.listdir(folder_path)

    # Regular expression pattern for American-style dates (MM-DD-YYYY)
    pattern = r'(\d{2})-(\d{2})-(\d{4})'

    for old_name in file_list:
        # Check if the file name matches the pattern
        match = re.search(pattern, old_name)
        if match:
            month, day, year = match.groups()

            # Format the new name with European-style date (DD-MM-YYYY)
            new_name = f"{day}-{month}-{year}{old_name[match.end():]}"

            # Create the full paths for old and new names
```

```
        old_path = os.path.join(folder_path, old_name)
        new_path = os.path.join(folder_path, new_name)

        # Rename the file
        os.rename(old_path, new_path)
        print(f"Renamed '{old_name}' to '{new_name}'")

if __name__ == "__main__":
    folder_path = "path_to_your_folder"  # Update with the path to your folder
    rename_files_with_dates(folder_path)
```

Sample Output

Let's assume you have three files in your folder that need renaming:
file1: report-08-15-2023.txt
file2: document-05-20-2022.docx
file3: notes-10-03-2021.txt
Here's how the program's output might look after running:
Renamed 'report-08-15-2023.txt' to '15-08-2023-report.txt'
Renamed 'document-05-20-2022.docx' to '20-05-2022-document.docx'
Renamed 'notes-10-03-2021.txt' to '03-10-2021-notes.txt'

The program identifies the American-style dates in the filenames and renames them to European-style dates. This output assumes that the program ran successfully without any errors. Remember to test the program on a small set of files before applying it to a larger dataset to ensure that it behaves as expected.

8. Backing Up a Folder into a ZIP File

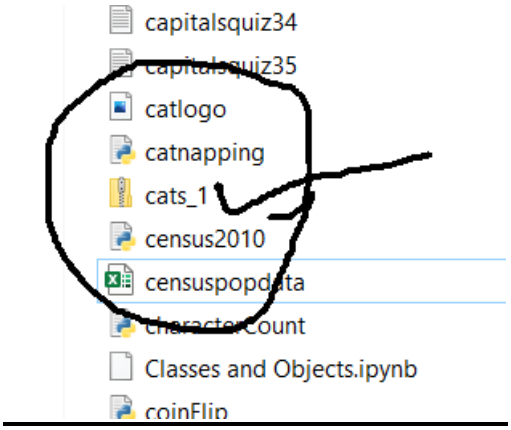
```
import zipfile, os
folder = input("Enter the folder name in the current working directory : ")
folder = os.path.abspath(folder) # make sure folder is absolute
number = 1
while True:
    zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
    if not os.path.exists(zipFilename):
        break
    number = number + 1

# Create the zip file.
print('Creating %s...' % (zipFilename))
backupZip = zipfile.ZipFile(zipFilename, 'w')
```

```
 # Walk the entire folder tree and compress the files in each folder.
 for foldername, subfolders, filenames in os.walk(folder):
    print('Adding files in %s...' % (foldername))
    # Add the current folder to the ZIP file.
    backupZip.write(foldername)
    # Add all the files in this folder to the ZIP file.
    for filename in filenames:
       if filename.startswith(os.path.basename(folder) + '_') and
 filename.endswith('.zip'):
          continue # don't backup the backup ZIP files
       backupZip.write(os.path.join(foldername, filename))
 backupZip.close()
print('Done.')
```

Sample Output

Enter the folder name in the current working directory: cats
Creating cats_1.zip...
Adding files in C:\Users\thyagu\18CS55\cats...
Done.

Compressed file in a given directory



9. Using Data Structures to Model Chess

In the realm of algebraic chess notation, the squares on the chessboard are denoted by a combination of numerical and alphabetical coordinates, exemplified in the diagram below:

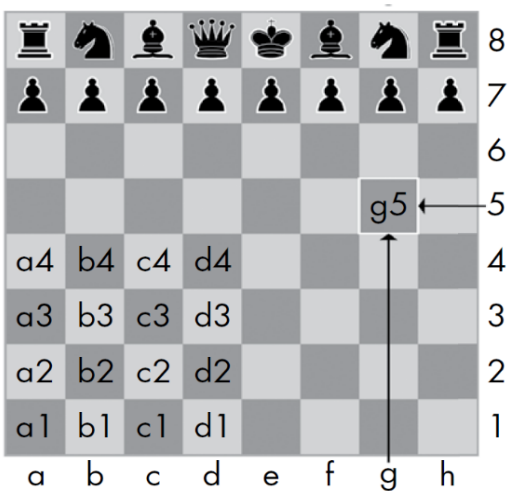


Image Source : [1] https://automatetheboringstuff.com/

Chess pieces are designated by specific letters: **K** for king, **Q** for queen, **R** for rook, **B** for bishop, and **N** for knight. Describing a move involves utilizing the letter of the piece along with the destination coordinates. A sequence of these moves portrays the events of a single turn (initiated by the white player). For example, annotation **2. Nf3 Nc6** signifies that, on the second turn of the game, the white player advanced a knight to **f3** and the black player moved a knight to **c6**.

Algebraic chess notation serves as a standardized system for documenting chess movements. This system empowers players to articulate their moves using a blend of letters and numbers that symbolize the squares on the chessboard.
The fundamental components of algebraic chess notation encompass:

K: King
Q: Queen
R: Rook
B: Bishop
N: Knight
No abbreviation for pawns

Square Designation

Each square on the chessboard is designated by a combination of a **letter** (column) and a **number** (row). The letters **a-h** represent the columns from **left to right**, and the numbers **1-8** represent the rows from **bottom to top**.

Move Representation

A move in algebraic notation typically consists of the piece abbreviation (if applicable), the destination square, and additional symbols to indicate the move type:

Pawn moves: Only the destination square is mentioned. If it's a capture, the source file is indicated as well.

Example: e4, exd5 (e4 pawn moves to e5, capturing d5 pawn)

Other pieces: The piece abbreviation is followed by the destination square. If multiple pieces of the same type can move to the same square, the source file or rank is mentioned to disambiguate.

Example: Nf3 (Knight moves to f3), R1e5 (Rook on the first rank moves to e5)

Special Moves

Castling: O-O for kingside castling and **O-O-O** for queenside castling.
En Passant: If a pawn captures **en** passant, the destination square is indicated and "**e.p.**" is added.

Example: exd6 e.p. (e5 pawn captures d6 pawn en passant)

Check and Checkmate

"+" is added after a move to indicate a check.
"#" is added after a move to indicate a checkmate.

Example: Qh5+ (Queen moves to **h5**, giving a check), **Qh5#** (Queen moves to **h5**, giving a checkmate)

Other Symbols

"=" is used to indicate pawn promotion. It is followed by the piece abbreviation to which the pawn promotes.

Example: e8=Q (Pawn on **e8** promotes to a Queen)

By using algebraic chess notation, players can easily record and communicate their moves. It is also commonly used in chess literature, annotations, and online platforms for game analysis.

10. Using Data Structures to Model Tic-Tac-Toe

The configuration of a tic-tac-toe board resembles a sizable hash symbol (#), featuring nine compartments, each capable of holding an X, an O, or remaining

unoccupied. When portraying the board utilizing a dictionary, a feasible approach involves assigning a unique string-based key to each slot, as depicted in the illustration below:

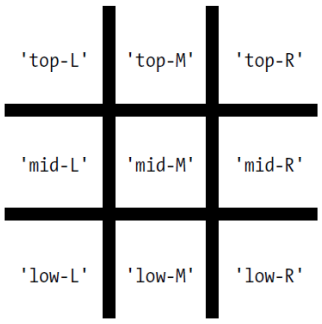


Image Source: [1] https://automatetheboringstuff.com/

Utilizing string representations, you can symbolize the contents of each slot on the board using '**X**', '**O**', or ' ' (a space character). Consequently, you will need to retain **nine** distinct strings. To achieve this, a suitable approach is to employ a dictionary comprising these string values. For instance, the string associated with the key '**top-R**' could indicate the top-right corner, the string linked to '**low-L**' could stand for the bottom-left corner, and similarly, the string correlated with '**mid-M**' could represent the middle section, and so forth. This dictionary constitutes a **data structure** that embodies the layout of a tic-tac-toe board. You can save this board-as-a-dictionary within a variable named **theBoard**.

```
theBoard =
{
        'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
        'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
        'low-L': ' ', 'low-M': ' ', 'low-R': ' '
}
```

The tic-tac-toe board depicted in the illustration below is represented by the data structure stored within the variable named **theBoard**:

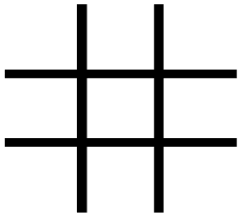


Figure 69.1 : Empty Tic Tac Toe Board

Given that each key's value within **theBoard** variable is a single-space string, this dictionary delineates an entirely vacant board. In a scenario where player X takes the initiative and selects the middle space as their move, you can use this dictionary to represent the resulting board configuration:

```
        theBoard =
{
        'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
        'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
         'low-L': ' ', 'low-M': ' ', 'low-R': ' '
}
```

The data structure contained within the **theBoard** variable now corresponds to the tic-tac-toe board illustrated in the figure below:

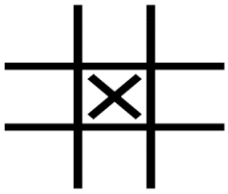


Figure 69.2: The First Move

An example of a board where player O emerges victorious by strategically placing **Os across the top** could resemble the following depiction:

```
theBoard =
{
        'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
        'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
        'low-L': ' ', 'low-M': ' ', 'low-R': 'X'
}
```

The data structure stored in the theBoard variable currently reflects the tic-tac-toe board illustrated in the figure below:

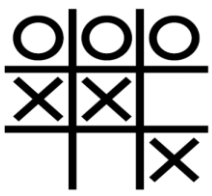


Figure 69.3: Player O wins

Python Program to simulate the working of Tic-Tac-Toe:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
          'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
          'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
   print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
   print('-+-+-')
   print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
   print('-+-+-')
   print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])

turn = 'X'

for i in range(9):
   printBoard(theBoard)
   print('Turn for ' + turn + '. Move on which space?')
   move = input()
   theBoard[move] = turn
   if turn == 'X':
      turn = 'O'
   else:
      turn = 'X'
printBoard(theBoard)
```

 # OutPut

```
| |
-+-+-
| |
-+-+-
| |
Turn for X. Move on which space?
mid-M
| |
-+-+-
|X|
-+-+-
| |
Turn for O. Move on which space?
low-L
```

```
| |
-+-+-
|X|
-+-+-
O| |
--snip--
O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

This isn't a complete **tic-tac-toe game**—for instance, it doesn't ever check whether a player has won—but it's enough to see **how data structures can be used in programs**.

70

References

1. www.python.org
2. https://chat.openai.com/
3. https://www.edureka.co/blog/python-programs/
4. Dr.Thyagaraju G S, “Introduction to Python Programming”, IIP Publishers, 2023
5. Al Sweigart, “Automate the Boring Stuff with Python”,1stEdition, No Starch Press, 2015. (Available under CC-BY-NC-SA license at https://automatetheboringstuff.com/)
6. https://www.learnbyexample.org/python-lambda-function/
7. Allen B. Downey, “Think Python: How to Think Like a Computer Scientist”, 2nd Edition, Green Tea Press, 2015. (Available under CC-BY-NC license at http://greenteapress.com/thinkpython2/thinkpython2.pdf
8. https://www.learnbyexample.org/python/
9. https://www.learnpython.org/
10. https://pythontutor.com/visualize.html#mode=edit
11. https://www.learnbyexample.org/python/
12. https://www.learnpython.org/
13. https://pythontutor.com/visualize.html#mode=edit
14. https://github.com/sushantkhara/Data-Structures-And-Algorithms-with-Python/raw/main/Python%203%20_%20400%20exercises%20and%20solutions%20for%20beginners.pdf
15. https://www.anaconda.com/

Book Resources

For more supplementary concepts, programs, question bank, old question papers (VTU), quiz, recent trends in Python and educational resources, kindly visit the author's website at:

https://tocxten.com/

ABOUT THE AUTHOR

Ms. Palguni G T is currently pursuing her B.E. in Computer Science and Business Systems at the esteemed Malnad College of Engineering (MCE) in Hassan. She displays a strong enthusiasm for acquiring knowledge in innovative areas related to Computer Science, including Artificial Intelligence, Machine Learning, and Data Science for Business Applications. Her passion lies in coding and application development, primarily utilizing the Python programming language, as well as various Web Technologies. She has certifications in Python to her credit. She presented a research paper titled "Emerging technologies for Business Applications - A Review", at a national conference on emerging technologies. The book titled "Python Quick Reference: Essential Syntax and Concepts" authored by Palguni G T has been crafted to cater to the needs of both beginners and experts. It serves as a concise review guide for placement interviews and university exams, making it a valuable resource for individuals at all skill levels.

ISBN: 978-93-5747-903-5

9 789357 479035

MRP Rs. 440/-

Selfypage Developers Pvt Ltd