UNITWISE NOTES

PYTHON

PROGRAMMING

PARTEEK BISHNOI

UNIT 1

Introduction To PYTHON

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.
Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions: **Python 2 and Python 3**. Both are quite different.

Writing our first program:

Just type in the following code after you start the interpreter.

```
print("Hello World")
```

Reason for increasing popularity

1. Emphasis on **code readability, shorter codes**, ease of writing
2. Programmers can express logical concepts in **fewer lines** of code in comparison to languages such as C++ or Java.
3. Python supports **multiple** programming paradigms, like object-oriented, imperative and functional programming or procedural.
4. There exists inbuilt functions for almost all of the frequently used concepts.
5. Philosophy is “Simplicity is the best”.

Softwares making use of Python
Python has been successfully embedded in a number of software products as a scripting language.

1. GNU Debugger uses Python as a **pretty printer** to show complex structures such as C++ containers.
2. Python has also been used in artificial intelligence
3. Python is often used for **natural language processing** tasks.

Current Applications of Python

1. A number of Linux distributions use installers written in Python example in Ubuntu we have the **Ubiquity**
2. Python has seen extensive use in the **information security industry**, including in exploit development.
3. Raspberry Pi– single board computer uses Python as its principal user-programming language.
4. Python is now being used **Game Development** areas also.

Pros:

1. Ease of use
2. Multi-paradigm Approach

Cons:

1. Slow speed of execution compared to C,C++
2. Absence from mobile computing and browsers
3. For the C,C++ programmers switching to python can be irritating as the language requires proper indentation of code. Certain variable names commonly used like sum are functions in python. So C, C++ programmers have to look out for these.

Advantages :

1. Presence of third-party modules
2. Extensive support libraries(NumPy for numerical calculations, Pandas for data analytics etc)
3. Open source and community development
4. Versatile, Easy to read, learn and write
5. User-friendly data structures
6. High-level language
7. Dynamically typed language(No need to mention data type based on the value assigned, it takes data type)
8. Object-oriented language
9. Portable and Interactive
10. Ideal for prototypes – provide more functionality with less coding
11. Highly Efficient(Python's clean object-oriented design provides enhanced process control, and the language is equipped with excellent text processing and integration capabilities, as well as its own unit testing framework, which makes it more efficient.)
12. (IoT)Internet of Things Opportunities
13. Interpreted Language
14. Portable across Operating systems

Applications :

1. GUI based desktop applications
2. Graphic design, image processing applications, Games, and Scientific/ computational Applications
3. Web frameworks and applications
4. Enterprise and Business applications
5. Operating Systems
6. Education
7. Database Access
8. Language Development
9. Prototyping
10. Software Development

Organizations using Python :

1. Google(Components of Google spider and Search Engine)
2. Yahoo(Maps)
3. YouTube
4. Mozilla
5. Dropbox
6. Microsoft
7. Cisco
8. Spotify
9. Quora

Python History and Versions

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, **Guido Van Rossum** published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language, which was capable of Exception Handling and interfacing with the Amoeba Operating System.
- The following programming languages influence Python:
    - ABC language.
    - Modula-3

Python Features

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. We have listed below a few essential features.

1) Easy to Learn and Use

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.

2) Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**. It will take only one line to execute, while Java or C takes multiple lines.

3) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

5) Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org. It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

6) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

8) Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

9) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

10) Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

11. Embeddable

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

12. Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to **x,** then we don't need to write **int x = 15.** Just write x = 15.

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Character Set

A character set in Python is a collection of legal characters that a scripting language will recognize when writing a script. We are referring to the Python coding language in this instance. Therefore, the character set in Python is a legitimate collection of characters that the Python language can recognize. These represent the Python scripting language's supported characters. Python is compatible with all ASCII and Unicode characters, including:

- **Alphabets:** These include all the small (a-z) and capital (A-Z) alphabets.
- **Digits:** It includes all the single digits 0-9.
- **Special Symbols:** It includes all the types of special characters," 'l ; : ! ~ @ # $ % ^ ` & * ( ) _ + - = { } [ ] \ .
- **White Spaces:** White spaces are also a part of the character set. These are tab space, newline, blank space, and carriage return.
- **Other:** Python supports all the types of ASCII and UNICODE characters that constitute the Python character set.

Tokens

The smallest distinct element in a Python program is called a token. Tokens are used to construct each phrase and command in a program.

Python Keywords

Every scripting language has designated words or keywords, with particular definitions and usage guidelines. Python is no exception. The fundamental constituent elements of any Python program are Python keywords.

This tutorial will give you a basic overview of all Python keywords and a detailed discussion of some important keywords that are frequently used.

Introducing Python Keywords

Python keywords are unique words reserved with defined meanings and functions that we can only apply for those functions. You'll never need to import any keyword into your program because they're permanently present.

Python's built-in methods and classes are not the same as the keywords. Built-in methods and classes are constantly present; however, they are not as limited in their application as keywords.

Assigning a particular meaning to Python keywords means you can't use them for other purposes in our code. You'll get a message of SyntaxError if you attempt to do the same. If you attempt to assign anything to a built-in method or type, you will not receive a SyntaxError message; however, it is still not a smart idea.

Python contains thirty-five keywords in the most recent version, i.e., Python 3.8. Here we have shown a complete list of Python keywords for the reader's reference.

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

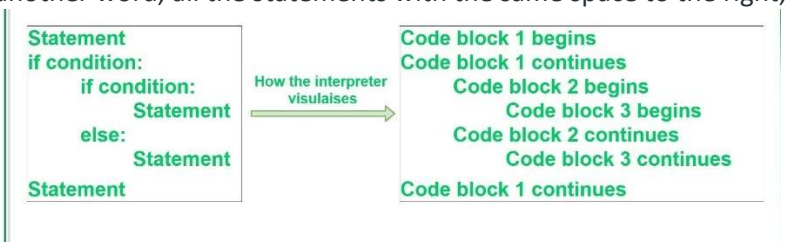
In distinct versions of Python, the preceding keywords might be changed. Some extras may be introduced, while others may be deleted. By writing the following statement into the coding window, you can anytime retrieve the collection of keywords in the version you are working on

Indentation in Python

Indentation is a very important concept of Python because without properly indenting the Python code, you will end up seeing IndentationError and the code will not get compiled.

Python Indentation

Python indentation refers to adding white space before a statement to a particular block of code. In another word, all the statements with the same space to the right, belong to the same code block.



Example of Python Indentation

- Statement (line 1), if condition (line 2), and statement (last line) belongs to the same block which means that after statement 1, if condition will be executed. and suppose the if condition becomes False then the Python will jump to the last statement for execution.
- The nested if-else belongs to block 2 which means that if nested if becomes False, then Python will execute the statements inside the else condition.
- Statements inside nested if-else belong to block 3 and only one statement will be executed depending on the if-else condition.

Python indentation is a way of telling a Python interpreter that the group of statements belongs to a particular block of code. A block is a combination of all these statements. Block can be regarded as the grouping of statements for a specific purpose. Most programming languages like C, C++, and Java use braces { } to define a block of code. Python uses indentation to highlight the blocks of code. Whitespace is used for indentation in Python. All statements with the same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right. You can understand it better by looking at the following lines of code.

Python Comments

We'll study how to write comments in our program in this article. We'll also learn about single-line comments, multi-line comments, documentation strings, and other Python comments.

Introduction to Python Comments

We may wish to describe the code we develop. We might wish to take notes of why a section of script functions, for instance. We leverage the remarks to accomplish this. Formulas, procedures, and sophisticated business logic are typically explained with comments. The Python interpreter overlooks the remarks and solely interprets the script when running a program. Single-line comments, multi-line comments, and documentation strings are the 3 types of comments in Python.

Advantages of Using Comments

Our code is more comprehensible when we use comments in it. It assists us in recalling why specific sections of code were created by making the program more understandable.

Aside from that, we can leverage comments to overlook specific code while evaluating other code sections. This simple technique stops some lines from running or creates a fast pseudo-code for the program.

Below are some of the most common uses for comments:

- Readability of the Code
- Restrict code execution
- Provide an overview of the program or project metadata
- To add resources to the code

Types of Comments in Python

In Python, there are 3 types of comments. They are described below:

Single-Line Comments

Single-line remarks in Python have shown to be effective for providing quick descriptions for parameters, function definitions, and expressions. A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation. Consider the accompanying code snippet, which shows how to use a single line comment:

The following is the comment:

```
# This code is to show an example of a single-line comment
```

The Python compiler ignores this line.

Multi-Line Comments

Python does not provide the facility for multi-line comments. However, there are indeed many ways to create multi-line comments.

With Multiple Hashtags (#)

In Python, we may use hashtags (#) multiple times to construct multiple lines of comments. Every line with a (#) before it will be regarded as a single-line comment.

Code

```
# it is a
# comment
# extending to multiple lines
```

In this case, each line is considered a comment, and they are all omitted.

Using String Literals

Because Python overlooks string expressions that aren't allocated to a variable, we can utilize them as comments.

Code

```
'it is a comment extending to multiple lines'
```

We can observe that on running this code, there will be no output; thus, we utilize the strings inside triple quotes(""") as multi-line comments.

Python Docstring

The strings enclosed in triple quotes that come immediately after the defined function are called Python docstring. It's designed to link documentation developed for Python modules, methods, classes, and functions together. It's placed just beneath the function, module, or class to explain what they perform. The docstring is then readily accessible in Python using the __doc__ attribute.

Code

```
# Code to show how we use docstrings in Python

def add(x, y):
    """This function adds the values of x and y"""
    return x + y

# Displaying the docstring of the add function
print( add.__doc__ )
```

Output:

```
This function adds the values of x and y
```

Command Line Arguments in Python

The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**. Python provides various ways of dealing with these types of arguments. The three most common are:

- Using sys.argv
- Using getopt module
- Using argparse module

Using sys.argv

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
One such variable is sys.argv which is a simple list structure. It's main purpose are:

- It is a list of command line arguments.
- len(sys.argv) provides the number of command line arguments.
- sys.argv[0] is the name of the current Python script.

Using getopt module

Python **getopt module** is similar to the getopt() function of C. Unlike sys module getopt module extends the separation of the input string by parameter validation. It allows both short, and long options including a value assignment. However, this module requires the use of the sys module to process input data properly. To use getopt module, it is required to remove the first element from the list of command-line arguments.

Syntax: *getopt.getopt(args, options, [long_options])*

Using argparse module

Using argparse module is a better option than the above two options as it provides a lot of options such as positional arguments, default value for arguments, help message, specifying data type of argument etc.

Note: As a default optional argument, it includes -h, along with its long version –help.

Python Operators

The operator is a symbol that performs a certain operation between two operands, according to one definition. In a particular programming language, operators serve as the foundation upon which logic is constructed in a programme. The different operators that Python offers are listed here.

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators
- Arithmetic Operators

Arithmetic Operators

Arithmetic operations between two operands are carried out using arithmetic operators. It includes the exponent (**) operator as well as the + (addition), - (subtraction), * (multiplication), / (divide), % (reminder), and // (floor division) operators.

Consider the following table for a detailed explanation of arithmetic operators.

Operator	Description
+ (Addition)	It is used to add two operands. For example, if a = 10, b = 10 => a+b = 20
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 5 => a - b = 15
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 4 => a * b = 80
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	As it calculates the first operand's power to the second operand, it is an exponent operator.
// (Floor division)	It provides the quotient's floor value, which is obtained by dividing the two operands.

Comparison operator

Comparison operators compare the values of the two operands and return a true or false Boolean value in accordance. The following table lists the comparison operators.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.

<=	The condition is met if the first operand is smaller than or equal to the second operand.
>=	The condition is met if the first operand is greater than or equal to the second operand.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

Assignment Operators

The right expression's value is assigned to the left operand using the assignment operators. The following table provides a description of the assignment operators.

Operator	Description
=	It assigns the value of the right expression to the left operand.
+=	By multiplying the value of the right operand by the value of the left operand, the left operand receives a changed value. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

Bitwise Operators

The two operands' values are processed bit by bit by the bitwise operators. Consider the case below.

Operator	Description
& (binary and)	A 1 is copied to the result if both bits in two operands at the same location are 1. If not, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.
^ (binary xor)	If the two bits are different, the outcome bit will be 1, else it will be 0.
~ (negation)	The operand's bits are calculated as their negations, so if one bit is 0, the next bit will be 1, and vice versa.
<< (left shift)	The number of bits in the right operand is multiplied by the leftward shift of the value of the left operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Logical Operators

The assessment of expressions to make decisions typically makes use of the logical operators. The following logical operators are supported by Python.

Operator	Description
and	The condition will also be true if the expression is true. If the two expressions a and b are the same, then a and b must both be true.
or	The condition will be true if one of the phrases is true. If a and b are the two expressions, then an or b must be true if and is true and b is false.
not	If an expression **a** is true, then not (a) will be false and vice versa.

/

Membership Operators

The membership of a value inside a Python data structure can be verified using Python membership operators. The result is true if the value is in the data structure; otherwise, it returns false.

Operator	Description
in	If the first operand cannot be found in the second operand, it is evaluated to be true (list, tuple, or dictionary).
not in	If the first operand is not present in the second operand, the evaluation is true (list, tuple, or dictionary).

Identity Operators

Operator	Description
is	If the references on both sides point to the same object, it is determined to be true.
is not	If the references on both sides do not point at the same object, it is determined to be true.

Operator Precedence

The order in which the operators are examined is crucial to understand since it tells us which operator needs to be considered first. Below is a list of the Python operators' precedence tables.

Operator	Description
**	Overall other operators employed in the expression, the exponent operator is given precedence.
~ + -	the minus, unary plus, and negation.
* / % //	the division of the floor, the modules, the division, and the multiplication.
+ -	Binary plus, and minus
>> <<	Left shift. and right shift
&	Binary and.

^ |	Binary xor, and or
<= < > >=	Comparison operators (less than, less than equal to, greater than, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Expressions in Python

An expression is a combination of operators and operands that is interpreted to produce some other value. In any programming language, an expression is evaluated as per the precedence of its operators. So that if there is more than one operator in an expression, their precedence decides which operation will be performed first. We have many different types of expressions in Python. Let's discuss all types along with some exemplar codes :

1. Constant Expressions: These are the expressions that have constant values only.

2. Arithmetic Expressions: An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis. The result of this type of expression is also a numeric value. The operators used in these expressions are arithmetic operators like addition, subtraction, etc. Here are some arithmetic operators in Python:

Operators	**Syntax**	**Functioning**
+	x + y	Addition
–	x – y	Subtraction
*	x * y	Multiplication

/	x / y	Division
//	x // y	Quotient
%	x % y	Remainder
**	x ** y	Exponentiation

3. Integral Expressions: These are the kind of expressions that produce only integer results after all computations and type conversions.

4. Floating Expressions: These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.

5. Relational Expressions: In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < , >= , <=). Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end. These expressions are also called Boolean expressions.

6. Logical Expressions: These are kinds of expressions that result in either *True* or *False.* It basically specifies one or more conditions. For example, (10 == 9) is a condition if 10 is equal to 9. As we know it is not correct, so it will return False. Studying logical expressions, we also come across some logical operators which can be seen in logical expressions most often. Here are some logical operators in Python:

Operator	**Syntax**	**Functioning**
and	P and Q	It returns true if both P and Q are true otherwise returns false
or	P or Q	It returns true if at least one of P and Q is true
not	not P	It returns true if condition P is false

7. Bitwise Expressions: These are the kind of expressions in which computations are performed at bit level.

8. Combinational Expressions: We can also use different types of expressions in a single expression, and that will be termed as combinational expressions.
But when we combine different types of expressions or use multiple operators in a single expression, operator precedence comes into play.

Multiple operators in expression (Operator Precedence)

It's a quite simple process to get the result of an expression if there is only one operator in an expression. But if there is more than one operator in an expression, it may give different results on basis of the order of operators executed. To sort out these confusions, the *operator precedence* is defined. Operator Precedence simply defines the priority of operators that which operator is to be executed first. Here we see the operator precedence in Python, where the operator higher in the list has more precedence or priority:

Precedence	Name	Operator
1	Parenthesis	( ) [ ] { }
2	Exponentiation	**
3	Unary plus or minus, complement	-a , +a , ~a
4	Multiply, Divide, Modulo	/ * // %
5	Addition & Subtraction	+ –
6	Shift Operators	>> <<
7	Bitwise AND	&
8	Bitwise XOR	^
9	Bitwise OR	|
10	Comparison Operators	>= <= > <
11	Equality Operators	== !=
12	Assignment Operators	= += -= /= *=

Precedence	Name	Operator
13	Identity and membership operators	is, is not, in, not in
14	Logical Operators	and, or, not

So, if we have more than one operator in an expression, it is evaluated as per operator precedence. For example, if we have the expression “10 + 3 * 4”. Going without precedence it could have given two different outputs 22 or 52. But now looking at operator precedence, it must yield 22.

Python print() Function

The python **print()** function prints the given object to the screen or other standard output devices.

Python print() function Example

```
print("Python is programming language.")
```

Output:

```
Python is programming language.
```

Python input() Function

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

Python input() Function Example

```
# Calling function
val = input("Enter a value: ")
# Displaying result
print("You entered:",val)
```

Output:

```
Enter a value: 45
You entered: 45
```

Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

Python eval() Function Example

```
1. x = 8
2. print(eval('x + 1'))
```

Output:

```
9
```

Python Data Types

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

```
1. a = 5
```

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

```
1. a=10
2. b="Hi Python"
3. c = 10.5
4. print(type(a))
5. print(type(b))
6. print(type(c))
```

Output:

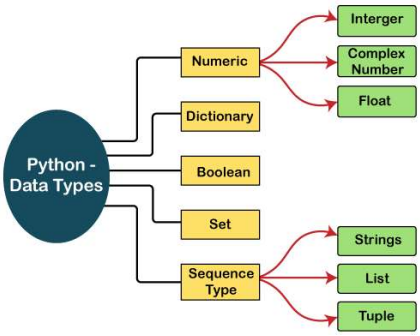
```
<type 'int'>
<type 'str'>
<type 'float'>
```

Standard data types

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. Numbers
2. Sequence Type
3. Boolean
4. Set
5. Dictionary



In this section of the tutorial, we will give a brief introduction of the above data-types. We will discuss each one of them in detail later in this tutorial.

Numbers

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python creates Number objects when a number is assigned to a variable.

Python supports three types of numeric data.

1. **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
3. **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

Python Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.

It is recommended to use lowercase letters for the variable name. Rahul and rahul both are two different variables.

Declaring Variable and Assigning Values

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

Python Variable Types

There are two types of variables in Python - Local variable and Global variable. Let's understand the following variables.

Local Variable

Local variables are the variables that declared inside the function and have scope within the function. Let's understand the following example.

Global Variables

Global variables can be used throughout the program, and its scope is in the entire program. We can use global variables inside or outside the function.

A variable declared outside the function is the global variable by default. Python provides the **global** keyword to use global variable inside the function. If we don't use the **global** keyword, the function treats it as a local variable.

Mutable and Immutable Data Types in Python

Mutable or immutable is the fancy word for explaining the property of data types of being able to get updated after being initialized. The basic explanation is thus: A mutable object is one whose internal state is changeable. On the contrary, once an immutable object in Python has been created, we cannot change it in any way.

What are Mutable Data Types?

Anything is said to be mutable when anything can be modified or changed. The term "mutable" in Python refers to an object's capacity to modify its values. These are frequently the things that hold a data collection.

What are Immutable Data Types?

Immutable refers to a state in which no change can occur over time. A Python object is referred to as immutable if we cannot change its value over time. The value of these Python objects is fixed once they are made.

List of Mutable and Immutable objects

Python mutable data types:

- Lists
- Dictionaries
- Sets
- User-Defined Classes (It depends on the user to define the characteristics of the classes)

Python immutable data types:

- Numbers (Integer, Float, Complex, Decimal, Rational & Booleans)
- Tuples
- Strings
- Frozen Sets

Mutable Objects	Immutable Objects
Objects can be changed after creation.	Objects can't be changed after creation.
Generally provides a method to add or remove elements.	It does not provide any method to add, remove or change the element.
Slower to access compared to immutable objects.	Quicker to access compared to mutable objects.
Changing mutable objects is easy and efficient.	Expensive or can't be changed.
E.g., Dictionary, Sets, Lists, etc.	E.g., Strings, Bytes, Frozen sets, tuples, etc.

Namespace

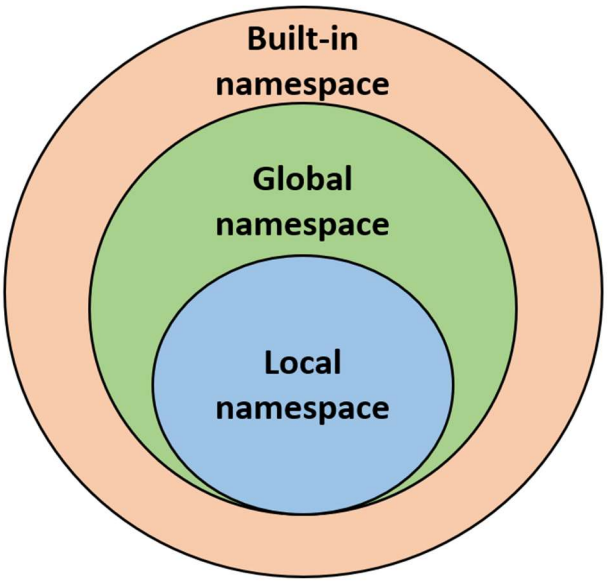
A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary. Let's go through an example, a directory-file system structure in computers. Needless to say, that one can have multiple directories having a file with the same name inside every directory. But one can get directed to the file, one wishes, just by specifying the absolute path to the file.

Real-time example, the role of a namespace is like a surname. One might not find a single "Alice" in the class there might be multiple "Alice" but when you particularly ask for "Alice Lee" or "Alice Clark" (with a surname), there will be only one (time being don't think of both first name and surname are same for multiple students).

On similar lines, the Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace. So, the division of the word itself gives a little more information. Its **Name** (which means name, a unique identifier) + **Space**(which talks something related to scope). Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.

Types of namespaces :

When Python interpreter runs solely without any user-defined modules, methods, classes, etc. Some functions like print(), id() are always present, these are built-in namespaces. When a user creates a module, a global namespace gets created, later the creation of local functions creates the local namespace. The **built-in namespace** encompasses the **global namespace** and the global namespace encompasses the **local namespace**.



Type of Namespaces

The lifetime of a namespace :

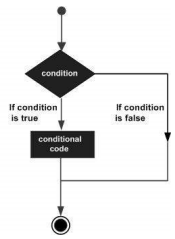
A lifetime of a namespace depends upon the scope of objects, if the scope of an object ends, the lifetime of that namespace comes to an end. Hence, it is not possible to access the inner namespace's objects from an outer namespace.

Decision statement in Python

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages −



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Boolean Type

In Python, the boolean type is a built-in data type that represents the truth values `True` and `False`. Booleans are used to perform logical operations, make comparisons, and control the flow of programs through conditional statements.

The boolean type has two possible values:

- `True`: Represents a true or affirmative condition.

- `False`: Represents a false or negative condition.

Here are some examples of using boolean values in Python:

```python

is_active = True

is_admin = False

print(is_active)  # Output: True

print(is_admin)  # Output: False

```

Boolean values can be assigned to variables or used directly in expressions. They are often the result of comparison operations or logical operations.

Boolean operators

Comparison Operators: Comparison operators are used to compare values and produce boolean results. Here are some examples:

```python
x = 5
y = 10
print(x == y)  # Output: False (equality check)
print(x < y)   # Output: True (less than check)
print(x >= y)  # Output: False (greater than or equal to check)
```

Logical Operators:

Logical operators are used to combine boolean values or expressions and produce boolean results. Here are the three logical operators in Python:

- `and`: Returns `True` if both operands are `True`.
- `or`: Returns `True` if at least one of the operands is `True`.
- `not`: Returns the opposite boolean value of the operand.

```python
is_logged_in = True
is_admin = False
print(is_logged_in and is_admin)  # Output: False
print(is_logged_in or is_admin)   # Output: True
print(not is_admin)               # Output: True
```

Boolean values are essential for controlling the flow of programs using conditional statements such as `if`, `while`, and `for` loops. They allow you to execute certain blocks of code based on whether a condition is true or false.

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

Boolean values can also be combined with conditional statements and logical operators to create more complex and flexible program logic.

Overall, boolean values and operations play a crucial role in decision making, flow control, and conditional execution in Python programs.

Python If-else statements

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

Indentation in Python

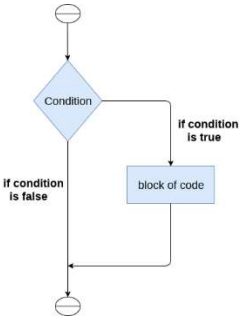
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

```
1. if expression:
2.     statement
```

Example 1

```
1. num = int(input("enter the number?"))
2. if num%2 == 0:
3.     print("Number is even")
```

Output:

```
enter the number?10
Number is even
```

Example 2 : Program to print the largest of the three numbers.

```
1. a = int(input("Enter a? "));
2. b = int(input("Enter b? "));
3. c = int(input("Enter c? "));
4. if a>b and a>c:
5.     print("a is largest");
6. if b>a and b>c:
7.     print("b is largest");
8. if c>a and c>b:
9.     print("c is largest");
```

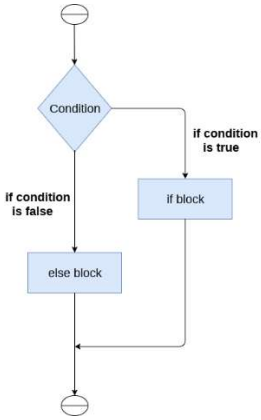
Output:

```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below.

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```

Example 1 : Program to check whether a person is eligible to vote or not.

```
age = int (input("Enter your age? "))
if age>=18:
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

Output:

```
Enter your age? 90
You are eligible to vote !!
```

Nested Conditionals Statements

Nested conditional statements, also known as nested if statements, are a way of including one or more if statements inside another if statement. This allows for more complex conditions and decision-making scenarios where multiple conditions need to be evaluated.

The syntax for a nested if statement in Python is as follows:

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

Here's an example to illustrate the usage of nested if statements:

```python
age = 20
country = "USA"
if country == "USA":
    if age >= 18:
        print("You are eligible to vote in the USA.")
    else:
        print("You are not eligible to vote in the USA.")
else:
    print("You are not in the USA.")
```

In this example, we have nested an if statement inside another if statement. The outer if statement checks if the `country` variable is equal to "USA". If it is, the nested if statement checks the `age` variable to determine if the person is eligible to vote. Depending on the conditions, the corresponding message is printed.

If the `country` is not "USA", the outer if statement's condition is false, and the program will execute the code inside the `else` block, printing "You are not in the USA."

It's important to maintain proper indentation to denote the nested structure of the if statements. Indentation in Python is typically four spaces, although you can use a tab or a different number of spaces, as long as it is consistent throughout your code.

Nested conditional statements allow for more granular control over the flow of your program and enable you to handle complex decision-making scenarios where multiple conditions need to be evaluated.

Multi-way Decision Statements

Multi-way decision statements, also known as multi-way if-elif-else statements, provide a way to evaluate multiple conditions and execute different blocks of code based on the outcome of these conditions. In Python, these statements allow you to handle various decision-making scenarios where you need to choose between multiple options.

The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax for a multi-way if-elif-else statement in Python is as follows:

```python
if condition1:
    # code to be executed if condition1 is true
elif condition2:
    # code to be executed if condition1 is false and condition2 is true
elif condition3:
    # code to be executed if condition1 and condition2 are false, and condition3 is true
else:
    # code to be executed if all conditions are false
```

Here's an example to illustrate the usage of a multi-way if-elif-else statement:

```python
score = 85
if score >= 90:
```

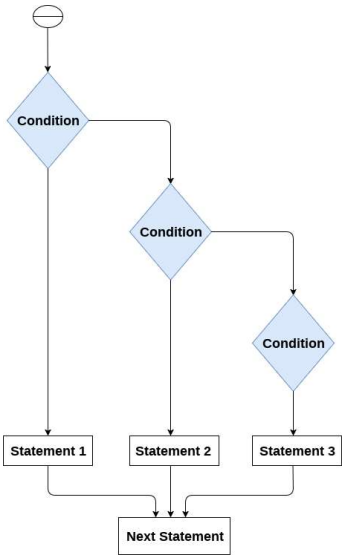
```
    print("A grade")
elif score >= 80:
    print("B grade")
elif score >= 70:
    print("C grade")
else:
    print("Below C grade")
```

In this example, the score is evaluated against multiple conditions. If the score is 90 or higher, the code block under the first `if` statement will be executed, and "A grade" will be printed. If the score is between 80 and 89, the code block under the `elif` statement will be executed, and "B grade" will be printed. Similarly, for scores between 70 and 79, "C grade" will be printed. If the score does not meet any of these conditions, the code block under the `else` statement will be executed, and "Below C grade" will be printed.

It's important to note that in a multi-way if-elif-else statement, only the code block corresponding to the first true condition encountered will be executed. Once a true condition is found, the rest of the conditions will be skipped.

You can have any number of `elif` statements between the initial `if` and the final `else`, allowing for multiple possible outcomes based on different conditions.

Multi-way decision statements are useful when you have multiple conditions to evaluate and need to choose one specific path of execution based on the true condition. They provide flexibility and allow you to handle various scenarios effectively.



Example 1

```
1. number = int(input("Enter the number?"))
2. if number==10:
3.     print("number is equals to 10")
4. elif number==50:
5.     print("number is equal to 50");
6. elif number==100:
7.     print("number is equal to 100");
8. else:
9.     print("number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

Example 2

```
1. marks = int(input("Enter the marks? "))
2. if marks > 85 and marks <= 100:
3.     print("Congrats ! you scored grade A ...")
4. elif marks > 60 and marks <= 85:
5.     print("You scored grade B + ...")
6. elif marks > 40 and marks <= 60:
7.     print("You scored grade B ...")
8. elif (marks > 30 and marks <= 40):
9.     print("You scored grade C ...")
10. else:
11.     print("Sorry you are fail ?")
```

UNIT 2

Loops and Control Statements

Python programming language provides the following types of loops to handle looping requirements.

Python While Loop

Until a specified criterion is true, a block of statements will be continuously executed in a Python while loop. And the line in the program that follows the loop is run when the condition changes to false.

Syntax of Python While

while expression:

statement(s)

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

- Python3

```
# prints Hello Geek 3 Times
count = 0
while (count < 3):
    count = count+1
    print("Hello Geek")
```

Output:

Hello Geek

Hello Geek

Hello Geek

See this for an example where a while loop is used for iterators. As mentioned in the article, it is not recommended to use a while loop for iterators in python.

Python range() function

The Python **range() function** returns a sequence of numbers, in a given range. The most common use of it is to iterate sequence on a sequence of numbers using Python loops.

Syntax of Python range() function

Syntax: *range(start, stop, step)*

Parameter:

- ***start****: [optional] start value of the sequence*
- ***stop:*** *next value after the end value of the sequence*

- ***step:*** *[optional] integer value, denoting the difference between any two numbers in the sequence.*

Return: *Returns a range type object.*

Example of Python range() function

- Python3

```
# print first 5 integers

# using python range() function

for i in range(5):

    print(i, end=" ")

print()
```

Output:

0 1 2 3 4

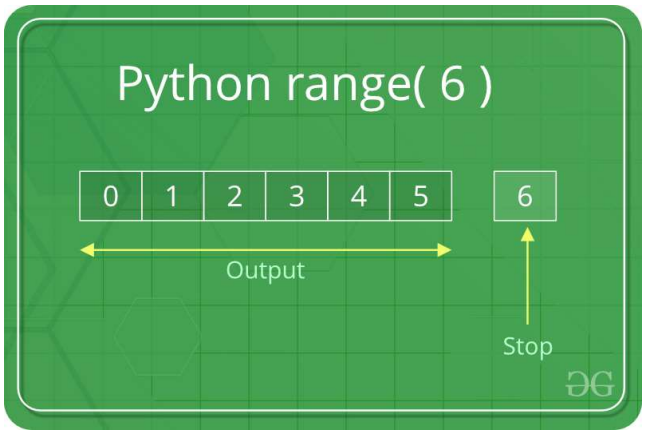
What is the use of the range function in Python

In simple terms, range() allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, the user can decide where that series of numbers will begin and end, as well as how big the difference will be between one number and the next. Python range() function takes can be initialized in 3 ways.

- range (stop) takes one argument.
- range (start, stop) takes two arguments.
- range (start, stop, step) takes three arguments.

Python range (stop)

When the user call range() with one argument, the user will get a series of numbers that starts at 0 and includes every whole number up to, but not including, the number that the user has provided as the stop.



Python range visualization

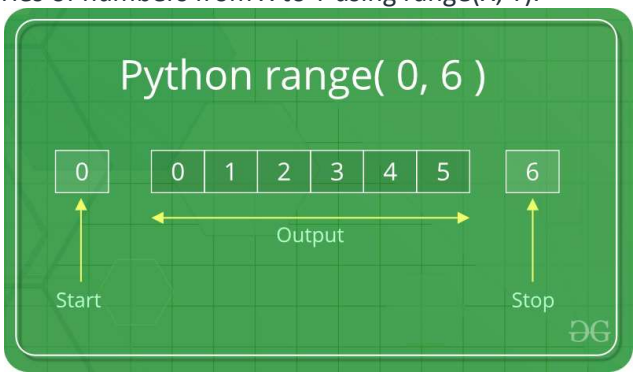
Example: Demonstration of Python range (stop)

- Python3

```
# printing first 6

# whole number

for i in range(6):

    print(i, end=" ")

print()
```

Output:
0 1 2 3 4 5

Python range (start, stop)
When the user call **range()** with two arguments, the user gets to decide not only where the series of numbers stops but also where it starts, so the user don't have to start at 0 all the time. Users can use range() to generate a series of numbers from X to Y using range(X, Y).



Python range visualization

Example: Demonstration of Python range (start, stop)

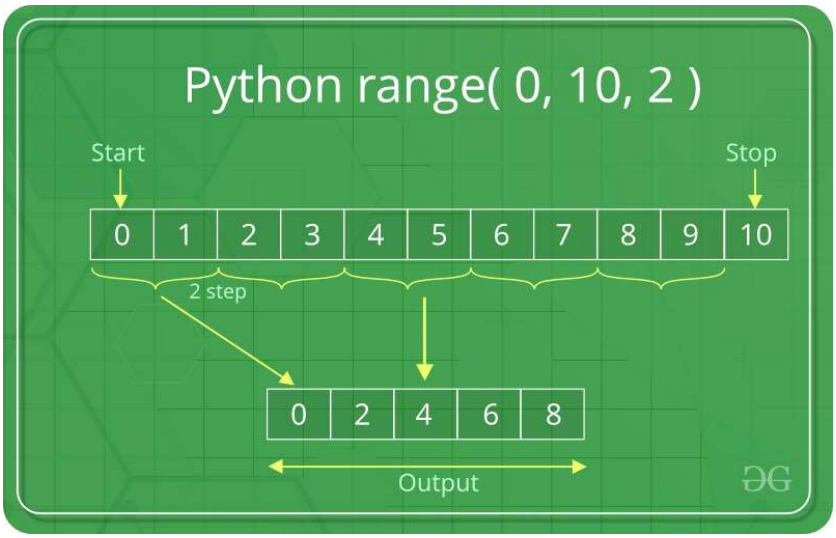
- Python3

```
# printing a natural

# number from 5 to 20

for i in range(5, 20):

    print(i, end=" ")
```

Output:
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Python range (start, stop, step)

When the user call range() with three arguments, the user can choose not only where the series of numbers will start and stop, but also how big the difference will be between one number and the next. If the user doesn't provide a step, then range() will automatically behave as if the step is 1. In this example, we are printing even numbers between 0 and 10, so we choose our starting point from 0(start = 0) and stop the series at 10(stop = 10). For printing an even number the difference between one number and the next must be 2 (step = 2) after providing a step we get the following output (0, 2, 4, 8).



Python range visualization

Example: Demonstration of Python range (start, stop, step)

- Python3

```
for i in range(0, 10, 2):
    print(i, end=" ")
print()
```

Output:

0 2 4 6 8

Some Important points to remember about the Python range() function:

- range() function only works with the integers, i.e. whole numbers.
- All arguments must be integers. Users can not pass a string or float number or any other type in a **start**, **stop** and **step** argument of a range().
- All three arguments can be positive or negative.
- The **step** value must not be zero. If a step is zero, python raises a ValueError exception.
- range() is a type in Python
- Users can access items in a range() by index, just as users do with a list:

Python for Loop

In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is a "**for in**" loop which is similar to for each loop in other languages.

Syntax:

for iterator_var in sequence:

    statements(s)

It can be used to iterate over iterators and a range.

- Python3

```
# Iterating over a list
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
    print(i)

# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("geeks", "for", "geeks")
for i in t:
    print(i)

# Iterating over a String
print("\nString Iteration")
s = "Geeks"
for i in s :
    print(i)

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
```

```
d['abc'] = 345

for i in d :

   print("%s %d" %(i, d[i]))
```

Output:

List Iteration

geeks

for

geeks

Tuple Iteration

geeks

for

geeks

String Iteration

G

e

e

k

s

Dictionary Iteration

xyz 123

abc 345

Time complexity: O(n), where n is the number of elements in the iterable (list, tuple, string, or dictionary).
Auxiliary space: O(1), as the space used by the program does not depend on the size of the iterable.
We can use a for-in loop for user-defined iterators. See this for example.

Python Nested Loops

Python programming language allows using one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax of Python Nested for Loop

The syntax for a nested for loop statement in Python programming language is as follows:

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
        statements(s)
```

Syntax of Python Nested while Loop

The syntax for a nested while loop statement in Python programming language is as follows:

```
while expression:
    while expression:
        statement(s)
        statement(s)
```

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

- Python3

```
from __future__ import print_function

for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

Output:

1

2 2

3 3 3

4 4 4 4

Infinite Loop

An infinite loop in Python is a loop that continues indefinitely without terminating. It executes repeatedly without any condition that would cause it to stop naturally. Infinite loops are usually created intentionally for specific purposes, such as running a server or continuously monitoring for events. However, they can also occur unintentionally due to logical errors in your code.

Here's an example of an intentional infinite loop using the `while` statement:

```python
while True:
    # code that runs indefinitely
    pass  # placeholder statement
```

In this example, the condition `True` is always true, so the loop will continue indefinitely. To exit an infinite loop like this, you can use a loop control statement such as `break` when a specific condition is met.

If you accidentally create an infinite loop and want to terminate it while running the code, you can usually do so by interrupting the program's execution. In most Python environments, you can press `Ctrl+C` on your keyboard to send an interrupt signal and stop the program.

It's important to be cautious when working with infinite loops to avoid unintentionally locking up your program or causing excessive resource usage.

Control Statements in Python

Loops are employed in Python to iterate over a section of code continually. Control statements are designed to serve the purpose of modifying a loop's execution from its default behaviour. Based on a condition, control statements are applied to alter how the loop executes. In this tutorial, we are covering every type of control statement that exists in Python.

Break Statements

In Python, the break statement is employed to end or remove the control from the loop that contains the statement. It is used to end nested loops (a loop inside another loop), which are shared with both types of Python loops. The inner loop is completed, and control is transferred to the following statement of the outside loop.

Code

```
1. # Python program to show how to control the flow of loops with the break statement
2.
3. Details = [[19, 'Itika', 'Jaipur'], [16, 'Aman', 'Bihar']]
4. for candidate in Details:
5.     age = candidate[0]
6.     if age <= 18:
7.         break
8.     print (f"{candidate[1]} of state {candidate[2]} is eligible to vote")
```

Output:

```
Itika of state Jaipur is eligible to vote
```

In the above code, if a candidate's age is less than or equal to 18, the interpreter won't generate the statement of eligibility. Otherwise, the interpreter will print a message mentioning that the candidate is eligible to vote in the console.

Continue Statements

When a Python interpreter sees a continue statement, it skips the present iteration's execution if the condition is satisfied. If the condition is not satisfied, it allows the implementation of the current iteration. It is employed to keep the program running even when it meets a break while being executed.

Code

```
# Python program to show how to control the flow of a loop using a continue statement
# Printing only the letters of the string
for l in 'I am a coder':
    if l == ' ':
        continue
    print ('Letter: ', l)
```

Output:

```
Letter:  I
Letter:  a
Letter:  m
Letter:  a
Letter:  c
Letter:  o
Letter:  d
Letter:  e
Letter:  r
```

In this code, when the if-statement encounters a space, the loop will continue to the following letter without printing the space.

Pass Statements

If the condition is met, the pass statement, or a null operator, is used by the coder to leave it as it is. Python's pass control statement changes nothing and moves on to the following iteration without stopping the execution or skipping any steps after completing the current iteration.

A coder can put the pass statement to prevent the interpreter from throwing an error when a loop or a code block is left empty.

Code

```
# Python program to show how to create empty code blocks using a pass statement
for l in 'Python':
    if l == 't':
        pass
    print('Letter: ', l)
```

Output:

```
Letter:  P
Letter:  y
Letter:  t
Letter:  h
Letter:  o
Letter:  n
```

Even if the condition was satisfied in the code above, as we can see, the pass statement had no effect, and execution went on to the subsequent iteration.

Introduction to Strings

A string is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it. Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

Example:
"Geeksforgeeks" or 'Geeksforgeeks'

Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

```
print("A Computer Science portal for geeks")
```

Output:
A Computer Science portal for geeks

String OPreations

Creating a String in Python
Strings in Python can be created using single quotes or double quotes or even triple quotes.

```
# Python Program for

# Creation of String

# Creating a String
```

```
# with single Quotes
String1 = 'Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)

# Creating a String
# with double Quotes
String1 = "I'm a Geek"
print("\nString with the use of Double Quotes: ")
print(String1)

# Creating a String
# with triple Quotes
String1 = '''I'm a Geek and I live in a world of "Geeks"'''
print("\nString with the use of Triple Quotes: ")
print(String1)

# Creating String with triple
# Quotes allows multiple lines
String1 = '''Geeks
        For
        Life'''
print("\nCreating a multiline String: ")
print(String1)
```

Output:

String with the use of Single Quotes:

Welcome to the Geeks World

String with the use of Double Quotes:

I'm a Geek

String with the use of Triple Quotes:

I'm a Geek and I live in a world of "Geeks"

Creating a multiline String:

Geeks

For

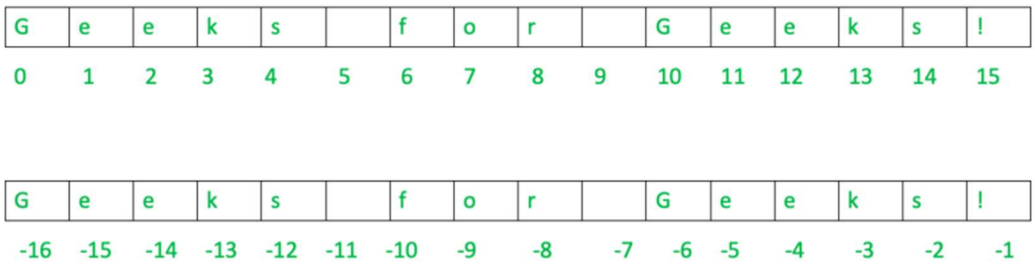
Life

Indexing

Indexing means referring to an element of an iterable by its position within the iterable. Each of a string's characters corresponds to an index number and each character can be accessed using its index number. We can access characters in a String in Two ways :

1. Accessing Characters by Positive Index Number
2. Accessing Characters by Negative Index Number

1. Accessing Characters by Positive Index Number: In this type of Indexing, we pass a Positive index(which we want to access) in square brackets. The index number starts from index number 0 (which denotes the first character of a string).



Indexing in Python

Example 1 (Positive Indexing) :

- python3

```
# declaring the string

str = "Geeks for Geeks !"
```

```
# accessing the character of str at 0th index
print(str[0])

# accessing the character of str at 6th index
print(str[6])

# accessing the character of str at 10th index
print(str[10])
```

Output

G

f

G

2. Accessing Characters by Negative Index Number: In this type of Indexing, we pass the Negative index(which we want to access) in square brackets. Here the index number starts from index number -1 (which denotes the last character of a string). **Example 2 (Negative Indexing) :**

- python3

```
# declaring the string
str = "Geeks for Geeks !"

# accessing the character of str at last index
print(str[-1])

# accessing the character of str at 5th index from the last
print(str[-5])

# accessing the character of str at 10th index from the last
print(str[-10])
```

Output

!

e

o

Reversing a Python String

With Accessing Characters from a string, we can also reverse them. We can Reverse a string by writing [::-1] and the string will be reversed.

- Python3

```
#Program to reverse a string

gfg = "geeksforgeeks"

print(gfg[::-1])
```

Output:

skeegrofskeeg

String Slicing

To access a range of characters in the String, the method of slicing is used. Slicing in a String is done by using a Slicing operator (colon).

- Python3

```
# Python Program to

# demonstrate String slicing


# Creating a String

String1 = "GeeksForGeeks"

print("Initial String: ")

print(String1)


# Printing 3rd to 12th character

print("\nSlicing characters from 3-12: ")

print(String1[3:12])


# Printing characters between

# 3rd and 2nd last character

print("\nSlicing characters between " +

    "3rd and 2nd last character: ")

print(String1[3:-2])
```

Output:

Initial String:

GeeksForGeeks

Slicing characters from 3-12:

ksForGeek

Slicing characters between 3rd and 2nd last character:

ksForGee

Slicing
Slicing in Python is a feature that enables accessing parts of the sequence. In slicing a string, we create a substring, which is essentially a string that exists within another string. We use slicing when we require a part of the string and not the complete string. **Syntax :**
string[start : end : step]

- *start : We provide the starting index.*
- *end : We provide the end index(this is not included in substring).*
- *step : It is an optional argument that determines the increment between each index for slicing.*

Example 1 :

- python3

```
# declaring the string

str ="Geeks for Geeks !"


# slicing using indexing sequence

print(str[: 3])

print(str[1 : 5 : 2])

print(str[-1 : -12 : -2])
```

Output

Gee

ek

!seGrf

Python Lists

Python Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In simple language, a list is a collection of things, enclosed in [ ] and separated by commas.

The list is a sequence data type which is used to store the collection of data. Tuples and String are other types of sequence data types.

Example of list in Python

Here we are creating Python **List** using [].

- Python3

```
Var = ["Geeks", "for", "Geeks"]

print(Var)
```

Output:

["Geeks", "for", "Geeks"]

Lists are the simplest containers that are an integral part of the Python language. Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

Operations on List

In Python, lists are versatile data structures that allow you to store and manipulate collections of items. Here are some common operations you can perform on lists:

1. Creating a list:

```python
my_list = [1, 2, 3, 4, 5]
```

2. Accessing elements:

- You can access individual elements in a list using indexing, starting from 0.

```python
my_list = [1, 2, 3, 4, 5]

print(my_list[0])  # Output: 1

print(my_list[2])  # Output: 3
```

3. Modifying elements:

- You can change the value of an element in a list by assigning a new value to a specific index.

```python
my_list = [1, 2, 3, 4, 5]
my_list[2] = 10
print(my_list)  # Output: [1, 2, 10, 4, 5]
```

4. List concatenation:

- You can concatenate two lists using the `+` operator.

```python
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2
print(result)  # Output: [1, 2, 3, 4, 5, 6]
```

5. List slicing:

- You can extract a sublist from a list using slicing, which allows you to specify a range of indices.

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

6. List length:

- You can get the length of a list using the `len()` function.

```python
```

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

length = len(my_list)

print(length)  # Output: 5
```

7. Adding elements to a list:

- You can add elements to a list using the `append()` method to add an element at the end.

```python
my_list = [1, 2, 3]

my_list.append(4)

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

- Alternatively, you can use the `extend()` method to append multiple elements from another list.

```python
my_list = [1, 2, 3]

my_list.extend([4, 5, 6])

print(my_list)  # Output: [1, 2, 3, 4, 5, 6]
```

8. Removing elements from a list:

- You can remove elements from a list using the `remove()` method by specifying the element to be removed.

```python
my_list = [1, 2, 3, 4, 5]

my_list.remove(3)

print(my_list)  # Output: [1, 2, 4, 5]
```

- If you know the index of the element, you can use the `del` statement or the `pop()` method.

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

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

9. Checking membership:

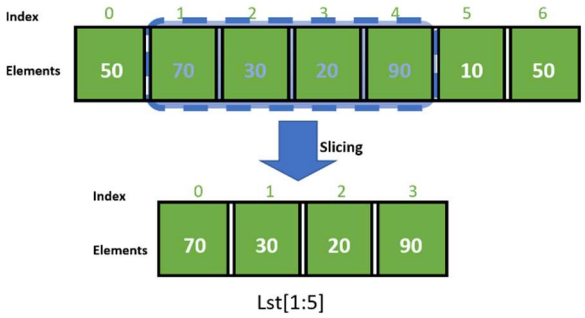
- You can check if an element is present in a list using the `in` keyword.

```python
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)  # Output: True
print(6 in my_list)  # Output: False
```

These are just some of the common operations you can perform on lists in Python. Lists provide a wide range of methods and functionalities, making them one of the most useful data structures in the language.

Slicing

As mentioned earlier list slicing is a common practice in Python and can be used both with positive indexes as well as negative indexes. The below diagram illustrates the technique of list slicing:



Lst[1:5]

The below program transforms the above illustration into python code:

- Python3

```
# Initialize list
Lst = [50, 70, 30, 20, 90, 10, 50]

# Display list
print(Lst[1:5])
```

Output:
[70, 30, 20, 90]

Inbuilt Functions for Lists

Python provides a wide range of built-in functions that can be used for list manipulation and operations. Here are some commonly used built-in functions for lists in Python:

1. `len()`: Returns the number of elements in a list.

```python
my_list = [1, 2, 3, 4, 5]
length = len(my_list)
print(length)  # Output: 5
```

2. `min()`: Returns the smallest element in a list.

```python
my_list = [5, 2, 9, 1, 7]
minimum = min(my_list)
print(minimum)  # Output: 1
```

3. `max()`: Returns the largest element in a list.

```python
my_list = [5, 2, 9, 1, 7]
maximum = max(my_list)
print(maximum)  # Output: 9
```

4. `sum()`: Returns the sum of all elements in a list.

```python
my_list = [1, 2, 3, 4, 5]
total = sum(my_list)
print(total)  # Output: 15
```

5. `sorted()`: Returns a new sorted list without modifying the original list.

```python
my_list = [3, 1, 4, 2, 5]
sorted_list = sorted(my_list)
print(sorted_list)  # Output: [1, 2, 3, 4, 5]
```

6. `reversed()`: Returns an iterator that yields the elements of a list in reverse order.

```python
my_list = [1, 2, 3, 4, 5]
reversed_list = list(reversed(my_list))
print(reversed_list)  # Output: [5, 4, 3, 2, 1]
```

7. `enumerate()`: Returns an iterator that provides both the index and value of each element in a list.

```python
my_list = ['apple', 'banana', 'cherry']
for index, value in enumerate(my_list):
    print(index, value)
# Output:
```

```
# 0 apple
# 1 banana
# 2 cherry
```

8. `zip()`: Combines multiple lists into a single list of tuples, where each tuple contains the corresponding elements from each list.

```python
numbers = [1, 2, 3]
fruits = ['apple', 'banana', 'cherry']
zipped = list(zip(numbers, fruits))
print(zipped)  # Output: [(1, 'apple'), (2, 'banana'), (3, 'cherry')]
```

These are just a few examples of the built-in functions available for lists in Python. These functions provide convenient ways to perform common operations on lists and can significantly simplify your code.

List Processing: Searching and Sorting

List processing often involves searching for specific elements or sorting the elements in a particular order. Here's how you can perform searching and sorting operations on lists in Python:

1. Searching for an element:

- You can search for an element in a list using various approaches, such as:

  - The `index()` method returns the index of the first occurrence of a specified element in the list.

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

  - The `in` keyword can be used to check if an element exists in the list.

  ```python
  my_list = [1, 2, 3, 4, 5]
  print(3 in my_list)  # Output: True
  ```

2. Sorting a list:

- To sort a list in ascending order, you can use the `sort()` method.

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

- If you want to sort a list in descending order, you can use the `sort()` method with the `reverse=True` parameter.

```python
my_list = [3, 1, 4, 2, 5]
my_list.sort(reverse=True)
print(my_list)  # Output: [5, 4, 3, 2, 1]
```

- Alternatively, you can use the `sorted()` function to create a new sorted list without modifying the original list.

```python
my_list = [3, 1, 4, 2, 5]
sorted_list = sorted(my_list)
print(sorted_list)  # Output: [1, 2, 3, 4, 5]
```

3. Custom sorting:

- You can perform custom sorting using the `sort()` method or the `sorted()` function by specifying a custom key or comparison function.

- For example, you can sort a list of strings based on the length of the strings.

```python
my_list = ["apple", "banana", "cherry", "date", "elderberry"]
```

```
my_list.sort(key=len)

print(my_list)  # Output: ['date', 'apple', 'cherry', 'banana', 'elderberry']
```

- Another example is sorting a list of tuples based on a specific element within the tuple.

```python
my_list = [(1, "apple"), (3, "banana"), (2, "cherry")]

my_list.sort(key=lambda x: x[0])  # Sort based on the first element of each tuple

print(my_list)  # Output: [(1, 'apple'), (2, 'cherry'), (3, 'banana')]
```

These operations provide you with the ability to search for specific elements in a list and sort the elements in various ways to meet your requirements.

Python Dictionary

Dictionary in Python is a collection of keys values, used to store data values like a map, which, unlike other data types which hold only a single value as an element.
Example of Dictionary in Python
Dictionary holds **key:value** pair. Key-Value is provided in the dictionary to make it more optimized.
Python3

```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}

print(Dict)
```

Output:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Need of dictionary

Dictionaries are an essential data structure in Python, providing a range of benefits and use cases. Here are some key reasons why dictionaries are needed:

1. Efficient data retrieval: Dictionaries allow for fast and efficient data retrieval based on keys. Instead of iterating through the entire collection, dictionaries use a hash table implementation that allows direct access to values based on their associated keys. This makes dictionaries ideal when you need to quickly access or retrieve data by using a unique identifier.

2. Key-value mapping: Dictionaries provide a flexible way to organize and represent data in a key-value format. This mapping allows you to associate values with specific keys, creating relationships and logical connections between different pieces of information. Keys can be of various types, including strings, integers, or even tuples, providing versatility in how you structure your data.

3. Data manipulation: Dictionaries offer efficient insertion, modification, and deletion operations. You can easily add new key-value pairs, update existing values, or remove entries as needed. This flexibility enables dynamic data management, allowing you to adapt and change your data structure as your program progresses.

4. Unordered collection: Dictionaries are unordered, meaning the elements are not stored in a specific order. This characteristic makes dictionaries well-suited for situations where the order of the data does not matter, and efficient data retrieval based on keys is more important. The lack of specific order allows for faster lookup times, regardless of the number of elements in the dictionary.

5. Unique keys: Dictionaries enforce unique keys, ensuring that each key corresponds to a single value. If you attempt to assign multiple values to the same key, the dictionary will store only the latest assigned value, replacing any previous values associated with that key. This uniqueness property helps maintain data integrity and prevents conflicts or inconsistencies in the mapping.

6. Use cases: Dictionaries are widely used in numerous scenarios, including:

- Storing and retrieving data based on identifiers or lookup keys, such as storing user information based on usernames or IDs.

- Representing and manipulating data from various sources, including JSON, databases, or API responses.

- Creating lookup tables, where keys correspond to specific values or actions.

- Implementing caches or memoization techniques to optimize the performance of computationally expensive operations.

- Storing configuration settings, preferences, or other types of metadata.

In summary, dictionaries are essential in Python due to their efficient data retrieval, flexible key-value mapping, rapid data manipulation, unordered collection nature, and unique key enforcement. They provide a versatile and powerful tool for managing and organizing data in a wide range of applications and programming tasks.

Operations on Directories:

In Python, there are several operations you can perform on directories (folders). Here are some commonly used operations on directories:

Creating a directory:

- You can create a new directory using the `os.mkdir()` function from the `os` module.

```python
import os
```

```
directory_path = "/path/to/new_directory"
os.mkdir(directory_path)
```

Addition to dictionary

To add or modify elements in a dictionary in Python, you can use the assignment operator (`=`) to assign values to specific keys. Here's how you can add or update elements in a dictionary:

1. Adding a new key-value pair:

```python
my_dict = {"name": "John", "age": 25}
my_dict["city"] = "New York"
print(my_dict)
# Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

2. Updating the value of an existing key:

```python
my_dict = {"name": "John", "age": 25}
my_dict["age"] = 30
print(my_dict)
# Output: {'name': 'John', 'age': 30}
```

3. Updating multiple key-value pairs at once using the `update()` method:

```python
my_dict = {"name": "John", "age": 25}
my_dict.update({"city": "New York", "age": 30})
print(my_dict)
# Output: {'name': 'John', 'age': 30, 'city': 'New York'}
```

4. Using the `setdefault()` method to add a key-value pair only if the key doesn't already exist:

```python
my_dict = {"name": "John", "age": 25}
```

```
my_dict.setdefault("city", "New York")
print(my_dict)
# Output: {'name': 'John', 'age': 25, 'city': 'New York'}

my_dict.setdefault("city", "London")  # Doesn't modify the value since 'city' already exists
print(my_dict)
# Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

Note: When adding or updating elements in a dictionary, the order of the elements is not guaranteed because dictionaries are unordered data structures.

By using these techniques, you can easily add new key-value pairs or update existing values in a dictionary in Python.

Retrieving Values

To retrieve values from a dictionary in Python, you can use the key associated with the value you want to access. Here are some ways to retrieve values from a dictionary:

1. Using square brackets (`[]`):

```python
my_dict = {"name": "John", "age": 25, "city": "New York"}
name = my_dict["name"]
age = my_dict["age"]
print(name)  # Output: John
print(age)  # Output: 25
```

2. Using the `get()` method:

```python
my_dict = {"name": "John", "age": 25, "city": "New York"}
name = my_dict.get("name")
age = my_dict.get("age")
print(name)  # Output: John
print(age)  # Output: 25
```

The `get()` method allows you to specify a default value that will be returned if the key does not exist in the dictionary:

```python
my_dict = {"name": "John", "age": 25, "city": "New York"}
occupation = my_dict.get("occupation", "Unknown")
print(occupation)  # Output: Unknown (since "occupation" key doesn't exist)
```

3. Using the `keys()` and `values()` methods:

- The `keys()` method returns a view object containing all the keys in the dictionary.
- The `values()` method returns a view object containing all the values in the dictionary.

```python
my_dict = {"name": "John", "age": 25, "city": "New York"}
keys = my_dict.keys()
values = my_dict.values()
print(keys)  # Output: dict_keys(['name', 'age', 'city'])
print(values)  # Output: dict_values(['John', 25, 'New York'])
```

You can convert these view objects into lists if needed:

```python
my_dict = {"name": "John", "age": 25, "city": "New York"}
keys = list(my_dict.keys())
values = list(my_dict.values())
print(keys)  # Output: ['name', 'age', 'city']
print(values)  # Output: ['John', 25, 'New York']
```

These methods allow you to retrieve values from a dictionary based on their corresponding keys. Choose the method that best suits your needs based on whether you want to access the value directly, handle missing keys, or obtain a view of all the keys or values in the dictionary.

Deleting or Removing a directory:

- You can remove an empty directory using the `os.rmdir()` function.

```python
import os
directory_path = "/path/to/directory"
os.rmdir(directory_path)
```

```

**Dictionary methods**

| Method | Description |
|---|---|
| dic.clear() | Remove all the elements from the dictionary |
| dict.copy() | Returns a copy of the dictionary |
| dict.get(key, default = “None”) | Returns the value of specified key |
| dict.items() | Returns a list containing a tuple for each key value pair |
| dict.keys() | Returns a list containing dictionary’s keys |
| dict.update(dict2) | Updates dictionary with specified key-value pairs |
| dict.values() | Returns a list of all the values of dictionary |
| pop() | Remove the element with specified key |
| popItem() | Removes the last inserted key-value pair |
| dict.setdefault(key,default= “None”) | set the key to the default value if the key is not specified in the dictionary |
| dict.has_key(key) | returns true if the dictionary contains the specified key. |
| dict.get(key, default = “None”) | used to get the value specified for the passed key. |
```

Python Tuples

Tuple is a collection of Python objects much like a list. The sequence of values stored in a tuple can be of any type, and they are indexed by integers.
Values of a tuple are syntactically separated by 'commas'. Although it is not necessary, it is more common to define a tuple by closing the sequence of values in parentheses. This helps in understanding the Python tuples more easily.

Operation on tuple

Creating a Tuple

In Python, tuples are created by placing a sequence of values separated by 'comma' with or without the use of parentheses for grouping the data sequence.

Note: Creation of Python tuple without the use of parentheses is known as Tuple Packing.

Creating a Tuple with Mixed Datatypes.

Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.). Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing 'comma' to make it a tuple.

Complexities for creating tuples:

Time complexity: O(1)
Auxiliary Space : O(n)

Accessing of Tuples

Tuples are immutable, and usually, they contain a sequence of heterogeneous elements that are accessed via unpacking or indexing (or even by attribute in the case of named tuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.
Note: In unpacking of tuple number of variables on the left-hand side should be equal to a number of values in given tuple a.

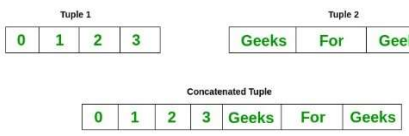
Complexities for accessing elements in tuples:

Time complexity: O(1)
Space complexity: O(1)

Concatenation of Tuples

Concatenation of tuple is the process of joining two or more Tuples. Concatenation is done by the use of '+' operator. Concatenation of tuples is done always from the end of the original tuple. Other arithmetic operations do not apply on Tuples.

Note- Only the same datatypes can be combined with concatenation, an error arises if a list and a tuple are combined.

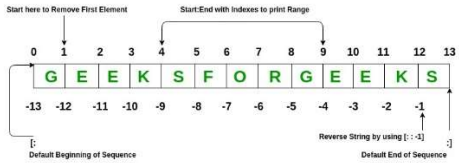


Time Complexity: O(1)
Auxiliary Space: O(1)

Slicing of Tuple

Slicing of a Tuple is done to fetch a specific range or slice of sub-elements from a Tuple. Slicing can also be done to lists and arrays. Indexing in a list results to fetching a single element whereas Slicing allows to fetch a set of elements.

Note- Negative Increment values can also be used to reverse the sequence of Tuples.



Complexities for traversal/searching elements in tuples:

Time complexity: O(1)
Space complexity: O(1)

Deleting a Tuple

Tuples are immutable and hence they do not allow deletion of a part of it. The entire tuple gets deleted by the use of del() method.

Note- Printing of Tuple after deletion results in an Error.

Built-In Methods

Built-in-Method	Description
index( )	Find in the tuple and returns the index of the given value where it's available
count( )	Returns the frequency of occurrence of a specified value

Built-In Functions

Built-in Function	Description
all()	Returns true if all element are true or if tuple is empty
any()	return true if any element of the tuple is true. if tuple is empty, return false
len()	Returns length of the tuple or size of the tuple
enumerate()	Returns enumerate object of tuple

Built-in Function	Description
max()	return maximum element of given tuple
min()	return minimum element of given tuple
sum()	Sums up the numbers in the tuple
sorted()	input elements in the tuple and return a new sorted list
tuple()	Convert an iterable to a tuple.

Tuples VS Lists:

Similarities	**Differences**
Functions that can be used for both lists and tuples: len(), max(), min(), sum(), any(), all(), sorted()	Methods that cannot be used for tuples: append(), insert(), remove(), pop(), clear(), sort(), reverse()
Methods that can be used for both lists and tuples: count(), Index()	we generally use 'tuples' for heterogeneous (different) data types and 'lists' for homogeneous (similar) data types.
Tuples can be stored in lists.	Iterating through a 'tuple' is faster than in a 'list'.
Lists can be stored in tuples.	'Lists' are mutable whereas 'tuples' are immutable.
Both 'tuples' and 'lists' can be nested.	Tuples that contain immutable elements can be used as a key for a dictionary.

Python Sets

In Python, a **Set** is an unordered collection of data types that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

Creating a Set

Sets can be created by using the built-in **set()** function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'.
Note: *A set cannot have mutable elements like a list or dictionary, as it is mutable.*

Time complexity: O(n), where n is the length of the input string or list.
Auxiliary space: O(n), where n is the length of the input string or list, since the size of the set created depends on the size of the input.
A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. Order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

Adding Elements to a Set

Using add() method
Elements can be added to the Set by using the built-in **add()** function. Only one element at a time can be added to the set by using add() method, loops are used to add multiple elements at a time with the use of add() method.
Note: *Lists cannot be added to a set as elements because Lists are not hashable whereas Tuples can be added because tuples are immutable and hence Hashable.*

Using update() method
For the addition of two or more elements Update() method is used. The update() method accepts lists, strings, tuples as well as other sets as its arguments. In all of these cases, duplicate elements are avoided.

Accessing a Set

Set items cannot be accessed by referring to an index, since sets are unordered the items has no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Removing elements from the Set

Using remove() method or discard() method:

Elements can be removed from the Set by using the built-in remove() function but a KeyError arises if the element doesn't exist in the set. To remove elements from a set without KeyError, use discard(), if the element doesn't exist in the set, it remains unchanged.

Using pop() method:

Pop() function can also be used to remove and return an element from the set, but it removes only the last element of the set.

Note: *If the set is unordered then there's no such way to determine which element is popped by using the pop() function.*

Using clear() method:

To remove all the elements from the set, clear() function is used.

Frozen sets in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.
If no parameters are passed, it returns an empty frozenset.

Advantages:

- **Unique Elements**: Sets can only contain unique elements, so they can be useful for removing duplicates from a collection of data.
- **Fast Membership Testing**: Sets are optimized for fast membership testing, so they can be useful for determining whether a value is in a collection or not.
- **Mathematical Set Operations:** Sets support mathematical set operations like union, intersection, and difference, which can be useful for working with sets of data.
- **Mutable**: Sets are mutable, which means that you can add or remove elements from a set after it has been created.

Disadvantages:

- **Unordered**: Sets are unordered, which means that you cannot rely on the order of the data in the set. This can make it difficult to access or process data in a specific order.
- **Limited Functionality:** Sets have limited functionality compared to lists, as they do not support methods like append() or pop(). This can make it more difficult to modify or manipulate data stored in a set.
- **Memory Usage:** Sets can consume more memory than lists, especially for small datasets. This is because each element in a set requires additional memory to store a hash value.
- **Less Commonly Used:** Sets are less commonly used than lists and dictionaries in Python, which means that there may be fewer resources or libraries available for working with them. This can make it more difficult to find solutions to problems or to get help with debugging.

Overall, sets can be a useful data structure in Python, especially for removing duplicates or for fast membership testing. However, their lack of ordering and limited functionality can also make them less versatile than lists or dictionaries, so it is important to carefully consider the advantages and disadvantages of using sets when deciding which data structure to use in your Python program.

Set Methods

Function	Description
add()	Adds an element to a set
remove()	Removes an element from a set. If the element is not present in the set, raise a KeyError

Function	Description
clear()	Removes all elements form a set
copy()	Returns a shallow copy of a set
pop()	Removes and returns an arbitrary set element. Raise KeyError if the set is empty
update()	Updates a set with the union of itself and others
union()	Returns the union of sets in a new set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another

Python Functions

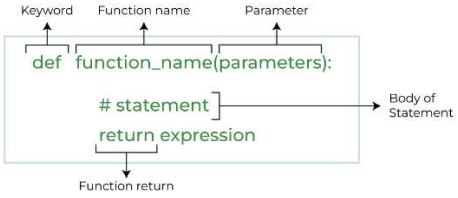
Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Some **Benefits of Using Functions**

- Increase Code Readability
- Increase Code Reusability

Python Function Declaration

The syntax to declare a function is:



Types of Functions in Python

There are mainly two types of functions in Python.

- **Built-in library function:** These are Standard functions in Python that are available to use.
- **User-defined function:** We can create our own functions based on our requirements.

Inbuilt function

Python provides a wide range of built-in functions that are readily available for use without requiring any additional libraries or installations. Here are some commonly used built-in functions in Python:

1. **print()**: Used to display output on the console.

   Example:

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

2. **input()**: Reads user input from the console.

   Example:

   ```python
   name = input("Enter your name: ")
   ```

3. **len()**: Returns the length (number of items) of an object.

   Example:

   ```python
   my_list = [1, 2, 3, 4, 5]
   length = len(my_list)
   ```

4. **type()**: Returns the type of an object.

   Example:

   ```python
   x = 10
   data_type = type(x)
   ```

5. **int()**: Converts a value to an integer data type.

   Example:

   ```python
   num_str = "10"
   num_int = int(num_str)
   ```

6. **str()**: Converts a value to a string data type.

   Example:

   ```python
   number = 10
   number_str = str(number)
   ```

7. **float()**: Converts a value to a floating-point data type.

   Example:

   ```python

   num_str = "3.14"

   num_float = float(num_str)

   ```

8. **range()**: Generates a sequence of numbers within a specified range.

   Example:

   ```python

   numbers = range(1, 6)  # generates numbers from 1 to 5

   ```

9. **sum()**: Returns the sum of all elements in an iterable.

   Example:

   ```python

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

   total = sum(my_list)

   ```

10. **max()** and **min()**: Returns the maximum and minimum values from an iterable, respectively.

    Example:

    ```python

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

    maximum = max(numbers)

    minimum = min(numbers)

    ```

Main Function

In Python, the `main()` function is not a built-in function but rather a convention used to indicate the starting point of a program. When a Python script is executed, the interpreter starts executing the code from the top of the file. However, sometimes you may want to define a specific section of code to be executed only when the script is run directly and not when it is imported as a module.

To achieve this, you can define a `main()` function and use the following construct:

```python
def main():
    # Your main code goes here
    print("This is the main function")

if __name__ == "__main__":
    main()
```

Here's how it works:

1. The `main()` function is defined to contain the main logic of your program.

2. The `if __name__ == "__main__":` statement checks whether the current script is the main script being executed.

3. If the script is being run directly (not imported), the `main()` function is called and the code inside it is executed.

This convention allows you to separate the reusable code into functions and have a clear entry point for your program.

By using the `main()` function and the `if __name__ == "__main__":` construct, you can ensure that certain code is executed only when the script is run directly, but not when it is imported as a module. This gives you more control and flexibility in organizing your code.

User defined functions

All the functions that are written by any us comes under the category of user defined functions. Below are the steps for writing user defined functions in Python.

- In Python, def keyword is used to declare user defined functions.
- An indented block of statements follows the function name and arguments which contains the body of the function.

Syntax:

```
def function_name():
    statements
    .
    .
```

Example:

- Python3

```
# Python program to
# demonstrate functions

# Declaring a function
def fun():
    print("Inside function")

# Driver's code
# Calling function
fun()
```

Output:

```
Inside function
```

Defining and Calling Function in python

In Python, you can define your own functions using the `def` keyword. Functions are blocks of code that perform a specific task and can be reused throughout your program. Here's the syntax for defining and calling a function in Python:

```python
def function_name(parameter1, parameter2, ...):
    # Code block for the function
    # You can perform operations, manipulate data, etc.
    # Optionally, you can return a value using the return statement
```

Call the function by using its name followed by parentheses and arguments (if any)

function_name(argument1, argument2, ...)

```

**Let's see an example:**

```python

def greet(name):

    # Function to greet the user

    print("Hello, " + name + "!")

# Call the greet() function

greet("Alice")  # Output: Hello, Alice!

greet("Bob")    # Output: Hello, Bob!

```

In this example, we defined a function called `greet()` that takes one parameter (`name`). Inside the function, we print a greeting message using the provided name. We then call the `greet()` function twice, passing different arguments each time.

Functions can also have return values. Here's an example:

```python

def add_numbers(a, b):

    # Function to add two numbers and return the result

    return a + b

# Call the add_numbers() function and store the result in a variable

result = add_numbers(3, 4)

print(result)  # Output: 7

```
```

In this case, the `add_numbers()` function takes two parameters (`a` and `b`) and returns their sum using the `return` statement. We call the function and store the result in the `result` variable, which we then print to the console.

You can define functions with multiple parameters, specify default values for parameters, use variable-length argument lists, and more. Python offers various features to make function definitions more flexible and powerful.

Parameter Passing

In Python, there are different ways to pass parameters to a function. The main ways of passing parameters are by value and by reference. Let's explore each of these methods:

1. **Passing Parameters by Value**: In Python, all function parameters are passed by object reference. However, for immutable objects like integers, strings, and tuples, the effect is similar to passing by value because the original object cannot be modified inside the function.

Example:

```python
def update_value(x):
    x = x + 1
    print("Inside the function:", x)
value = 5
update_value(value)
print("Outside the function:", value)
```

Output:

```
Inside the function: 6
Outside the function: 5
```

In this example, the `update_value()` function takes a parameter `x` and increments its value by 1. However, the original `value` variable remains unchanged outside the function because integers are immutable.

2. **Passing Parameters by Reference**: For mutable objects like lists and dictionaries, modifications made to the object inside the function are reflected outside the function.

Example:

```python
def update_list(lst):
    lst.append(4)
    print("Inside the function:", lst)
my_list = [1, 2, 3]
update_list(my_list)
print("Outside the function:", my_list)
```

Output:

```
Inside the function: [1, 2, 3, 4]
Outside the function: [1, 2, 3, 4]
```

In this example, the `update_list()` function takes a list `lst` and appends the value 4 to it. The modification made inside the function is reflected in the `my_list` variable outside the function.

It's important to note that even though all parameters in Python are passed by object reference, the behavior can be similar to passing by value for immutable objects because the function works with a copy of the reference. However, for mutable objects, modifications made inside the function affect the original object since they both refer to the same memory location.

If you want to avoid modifying the original object passed as a parameter, you can create a copy of the object inside the function using the appropriate methods or operators, such as `copy.deepcopy()` for nested objects.

It's also worth mentioning that Python doesn't support passing parameters strictly by reference or strictly by value, as some other programming languages do. The behavior is a combination of both, based on whether the object is mutable or immutable.

Actual and Formal Parameters

In Python, when defining a function, you can specify parameters that act as placeholders for the values that will be passed to the function when it is called. These parameters are often referred to as formal parameters. When the function is called, the values that are passed to it are called actual parameters or arguments.

Here's an example to illustrate the concept:

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

In this case, `name` is the formal parameter of the `greet()` function. When you call the function and pass an argument, that argument becomes the actual parameter.

```python
greet("Alice")  # "Alice" is the actual parameter
greet("Bob")    # "Bob" is the actual parameter
```

In the example above, "Alice" and "Bob" are the actual parameters passed to the `greet()` function. Inside the function, the formal parameter `name` takes the value of the corresponding actual parameter during the function call.

It's important to note that the number and order of actual parameters must match the number and order of formal parameters defined in the function. If there is a mismatch, Python will raise a `TypeError` or `ValueError` indicating the incorrect number of arguments or argument types.

Additionally, Python provides some flexibility in how you can pass the actual parameters. There are three main ways to pass arguments to a function:

1. **Positional arguments**: Arguments are passed based on their position and order. The first argument corresponds to the first formal parameter, the second argument to the second formal parameter, and so on.

2. **Keyword arguments**: Arguments are passed with their corresponding parameter names, allowing you to specify the values explicitly and out of order.

3. **Default arguments**: Parameters can have default values assigned to them. If an argument is not provided for a parameter with a default value, the default value is used instead.

Here's an example that demonstrates these argument-passing methods:

```python
def person_details(name, age, country=""):
    print("Name:", name)
    print("Age:", age)
    print("Country:", country)

person_details("Alice", 25)              # Positional arguments
person_details(age=30, name="Bob")       # Keyword arguments
person_details("Charlie", 40, "USA")     # Positional and keyword arguments
```

In this example, the `person_details()` function takes three parameters: `name`, `age`, and `country` (with a default value). You can call the function using positional arguments, keyword arguments, or a combination of both.

Understanding the distinction between formal parameters and actual parameters helps in correctly defining and calling functions while passing the appropriate values.

Default Parameter

In Python, you can assign default values to function parameters. Default parameters allow you to specify a value that will be used if no argument is provided for that parameter when calling the function. This provides flexibility and allows you to define functions that can be called with or without specific arguments.

Here's an example that demonstrates the usage of default parameters:

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

# Calling the function with both parameters
greet("Alice", "Hi")  # Output: Hi, Alice!

# Calling the function without providing the 'message' parameter
```

```
greet("Bob")  # Output: Hello, Bob!
```

In the example above, the `greet()` function has two parameters: `name` and `message`. The `message` parameter has a default value of "Hello". When calling the function, you can provide both parameters or just the `name` parameter. If the `message` argument is not provided, the default value "Hello" will be used.

When defining a function with default parameters, it's important to keep in mind the following:

- Default parameters should be placed at the end of the parameter list. For example, `def my_function(a, b, c=default_value)` is valid, but `def my_function(a=default_value, b, c)` is not.

- The default value is evaluated only once, at the time of function definition. This means that if the default value is a mutable object (e.g., a list or dictionary), modifications made to it will persist across multiple function calls.

Default parameters provide flexibility in function calls by allowing you to define sensible default values. They are particularly useful when you want to provide optional arguments or make certain parameters more convenient to use.

Global and Local Variables

In Python, variables can have either global or local scope, which determines where the variable can be accessed and used within a program.

Global Variables:

A global variable is defined outside of any function and can be accessed from anywhere within the program, including inside functions.

```python
global_var = 10  # Global variable

def my_function():
    print(global_var)  # Accessing global variable inside function

my_function()  # Output: 10

print(global_var)  # Output: 10
```

In the example above, `global_var` is a global variable that is accessible both inside and outside the `my_function()` function.

Local Variables:

A local variable is defined inside a function and can only be accessed within that function. It is not accessible outside of the function.

```python

def my_function():

    local_var = 20  # Local variable

    print(local_var)



my_function()  # Output: 20

print(local_var)  # Error: NameError: name 'local_var' is not defined

```

In this example, `local_var` is a local variable that is defined within the `my_function()` function. It can only be accessed within the function's scope and is not accessible outside of it.

It's important to note that if a local variable has the same name as a global variable, the local variable will take precedence within the function's scope.

```python

my_var = 5  # Global variable

def my_function():

    my_var = 10  # Local variable with the same name as the global variable

    print(my_var)  # Output: 10

my_function()  # Output: 10

print(my_var)  # Output: 5 (global variable remains unchanged)

```

In this case, `my_function()` has a local variable `my_var` that shadows the global variable `my_var` with the same name. Inside the function, the local variable is used, but outside the function, the global variable retains its original value.

It's important to understand the scoping rules to avoid confusion and unintended behavior when using global and local variables. Global variables are accessible from anywhere within the program, while local variables are confined to their respective function scopes.

Recursion

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, simpler instances. In other words, a recursive function solves a problem by solving smaller instances of the same problem until a base case is reached.

Here's an example of a recursive function to calculate the factorial of a number:

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

In this example, the `factorial()` function calculates the factorial of a given number `n`. The base case is when `n` is equal to 0, in which case the function returns 1. Otherwise, the function calls itself with `n - 1` as the argument and multiplies it by `n`. This process continues recursively until the base case is reached.

Let's see how we can use the `factorial()` function:

```python
result = factorial(5)
print(result)  # Output: 120
```

In this case, the `factorial(5)` call will compute the factorial of 5 by recursively multiplying 5 with the factorial of 4, then 4 with the factorial of 3, and so on until it reaches the base case of 0.

Recursion can be a powerful technique for solving problems that can be broken down into simpler, repetitive subproblems. However, it's essential to design recursive functions carefully to ensure they have well-defined base cases and make progress towards reaching those base cases. Otherwise, the recursive function may result in infinite recursion and lead to a stack overflow error.

It's worth noting that some problems can be solved more efficiently using iterative approaches rather than recursion. However, recursion can provide elegant and concise solutions for certain problems, particularly those that exhibit a recursive structure.

Passing Functions as Data

In Python, functions are considered first-class citizens, which means they can be treated like any other data type. This allows you to pass functions as arguments to other functions, return functions from functions, and assign functions to variables. This concept is often referred to as "passing functions as data" or "higher-order functions."

Here's an example that demonstrates passing a function as an argument to another function:

```python
def apply_operation(operation, a, b):
    return operation(a, b)

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

def multiply(x, y):
    return x * y

result1 = apply_operation(add, 3, 4)

print(result1)  # Output: 7

result2 = apply_operation(multiply, 3, 4)

print(result2)  # Output: 12
```

In this example, we have three functions: `apply_operation()`, `add()`, and `multiply()`. The `apply_operation()` function takes three arguments: `operation`, `a`, and `b`. The `operation` argument is a function that takes two arguments `a` and `b` and performs a specific operation. The `apply_operation()` function then calls the `operation` function with the given `a` and `b` arguments and returns the result.

We can pass different functions (`add` and `multiply`) as the `operation` argument to the `apply_operation()` function, resulting in different operations being performed on `a` and `b`.

In addition to passing functions as arguments, you can also return functions from other functions. Here's an example:

```python

def get_multiplier(factor):

    def multiply(x):

        return x * factor

    return multiply

multiply_by_2 = get_multiplier(2)

result = multiply_by_2(5)

print(result)  # Output: 10

```

In this example, the `get_multiplier()` function returns another function, `multiply()`, which multiplies a given number `x` by the `factor` provided to `get_multiplier()`. We assign the returned function to the `multiply_by_2` variable and then call it with an argument of 5, resulting in the multiplication of 5 by 2.

This capability of passing functions as data allows for dynamic behavior in Python programs, enabling you to create more flexible and reusable code. It is particularly useful in scenarios such as callbacks, event handling, and functional programming paradigms.

Lamda Function

In Python, a lambda function, also known as an anonymous function, is a way to create small, one-line functions without explicitly defining a function using the `def` keyword. Lambda functions are typically used for simple and concise operations.

The general syntax of a lambda function is as follows:

```python

lambda arguments: expression

```

Here's an example that demonstrates the use of a lambda function:

```python

addition = lambda x, y: x + y
```

result = addition(3, 4)

print(result) # Output: 7

```

In this example, we define a lambda function `addition` that takes two arguments `x` and `y` and returns their sum. We then call the lambda function with arguments `3` and `4`, and store the result in the `result` variable. The output is `7`.

Lambda functions can take any number of arguments, but they are typically used for simple operations. Here's another example that uses a lambda function to perform a calculation on a list of numbers:

```python

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

squared_numbers = list(map(lambda x: x ** 2, numbers))

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

```

In this example, we use the `map()` function along with a lambda function to square each element in the `numbers` list. The lambda function `lambda x: x ** 2` takes a single argument `x` and returns its square. The `map()` function applies this lambda function to each element in the `numbers` list, resulting in a new list `squared_numbers` containing the squared values.

Lambda functions are particularly useful in scenarios where a small, one-time function is needed, especially when passing a function as an argument to other functions (e.g., sorting, filtering, or reducing data). However, for more complex operations or functions that require multiple statements or control flow, it is generally recommended to use regular named functions defined with the `def` keyword for better readability and maintainability.

Modules

In Python, a module is a file containing Python code that defines variables, functions, and classes that can be used in other Python programs. Modules provide a way to organize code into reusable and logical units, making it easier to manage and maintain large projects.

To use code from a module, you need to import it into your program. Python provides several ways to import modules:
```

1. **Import the whole module**: You can import the entire module using the `import` statement. This allows you to access the module's contents using the module name as a prefix.

```python
import math

result = math.sqrt(16)

print(result)  # Output: 4.0
```

In this example, the `math` module is imported, and the `sqrt()` function from the module is used to calculate the square root of 16.

2. **Import specific names from a module**: If you only need specific functions or variables from a module, you can import them directly using the `from` keyword.

```python
from math import sqrt, pi

result = sqrt(16)

print(result)  # Output: 4.0

circumference = 2 * pi * 5

print(circumference)  # Output: 31.41592653589793
```

In this case, only the `sqrt()` function and the `pi` variable are imported from the `math` module. You can use them directly without the need to prefix them with the module name.

3. **Import the whole module with an alias**: You can import a module and give it an alias using the `as` keyword. This is useful when the module name is long or could cause conflicts with other names in your code.

```python
import math as m

result = m.sqrt(16)

print(result)  # Output: 4.0
```

```

In this example, the `math` module is imported with the alias `m`. This allows you to access the module's contents using `m` as a prefix instead of `math`.

Python comes with a standard library that provides a wide range of modules for various purposes. Additionally, there are numerous third-party modules available that can be installed using package managers like `pip`. These modules extend the functionality of Python and provide additional features and tools.

To create your own module, you can simply create a new Python file and define variables, functions, or classes within it. Then, you can import and use that module in other programs as described above.

Modules are an essential aspect of Python programming as they facilitate code reuse, organization, and modular development. They allow you to build complex programs by leveraging pre-existing functionality and promote good programming practices such as separation of concerns and modularity.

## Importing Own Module

To import your own module into another Python program, follow these steps:

1. Create a Python file with the code that defines the variables, functions, or classes you want to use as a module. Save the file with a `.py` extension.

**my_module.py**

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

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

2. In your main Python program, use the `import` statement to import your module.

main.py

```python
import my_module
```
```

```
my_module.greet("Alice")

result = my_module.add_numbers(3, 4)

print(result)  # Output: 7
```

In this example, the `my_module` module is imported using the `import` statement. You can access the functions from the module by prefixing them with the module name (`my_module`).

3. Run the main Python program, and it will utilize the code from your module.

```
Hello, Alice!

7
```

Note that the module file (`my_module.py`) should be in the same directory as the main Python program (`main.py`) or in a directory listed in the Python module search path.

You can also import specific functions or variables from your module using the `from` statement.

main.py

```python
from my_module import greet, add_numbers

greet("Bob")

result = add_numbers(5, 6)

print(result)  # Output: 11
```

In this case, the `greet()` and `add_numbers()` functions are imported directly from the `my_module` module. You can use them without prefixing them with the module name.

By organizing your code into modules, you can create reusable components and promote code maintainability and reusability. Make sure to choose meaningful names for your modules and avoid naming conflicts with Python's built-in modules or third-party modules to ensure smooth importing and usage.

Packages

In Python, a package is a way to organize related modules into a hierarchical directory structure. It provides a means to create a collection of modules that can be easily managed and distributed as a single unit. Packages allow for better organization, code reusability, and maintainability in larger Python projects.

Here's an example of a package structure:

```
my_package/
    __init__.py
    module1.py
    module2.py
    subpackage/
        __init__.py
        module3.py
```

In this example, we have a package named `my_package` that contains multiple modules. The package is represented by a directory, and each module is a separate Python file within that directory. The `__init__.py` files are empty files that indicate that the directories are Python packages.

To use the modules from a package, you need to import them using the package name and module name separated by a dot.

```python
import my_package.module1
import my_package.subpackage.module3
my_package.module1.some_function()
my_package.subpackage.module3.another_function()
```

Alternatively, you can use the `from ... import ...` statement to import specific modules from the package:

```python

from my_package import module1

from my_package.subpackage import module3

module1.some_function()

module3.another_function()

```

This allows you to use the functions, classes, or variables defined in the imported modules.

Packages can have multiple levels of nesting, allowing for a hierarchical structure. Subpackages are created by adding subdirectories within the package directory, each containing its own `__init__.py` file.

It's worth noting that packages often include an `__init__.py` file. This file can contain initialization code for the package or be left empty. The presence of this file indicates that the directory is a package.

Packages play a crucial role in organizing and structuring large Python projects, allowing for modular development, code sharing, and separation of concerns. By grouping related modules within packages, you can create a more maintainable and scalable codebase.

UNIT 3

Operations on File

Python provides several built-in functions and modules for performing operations on files. Here are some common operations you can perform on files in Python:

1. Opening a File:

To open a file, you can use the `open()` function. It takes the file path and mode as arguments. The mode can be 'r' for reading, 'w' for writing, 'a' for appending, or 'x' for creating a new file. Here's an example:

```python

file = open("file.txt", "r")  # Opens file.txt in read mode

```

2. Reading from a File:

To read the contents of a file, you can use the `read()` or `readlines()` methods. `read()` reads the entire file as a single string, while `readlines()` returns a list of lines. Here's an example:

```python

content = file.read()      # Reads the entire file

lines = file.readlines()   # Reads the file line by line

```

3. Writing to a File:

To write to a file, you can use the `write()` method. It takes a string as an argument and writes it to the file. If the file doesn't exist, it will be created. If it already exists, the previous contents will be overwritten. Here's an example:

```python

file.write("Hello, world!")   # Writes "Hello, world!" to the file

```

4. Appending to a File:

To append content to an existing file, you can open the file in append mode ('a') and then use the `write()` method to add content to the end of the file. Here's an example:

```python

file = open("file.txt", "a")  # Opens file.txt in append mode

file.write("Appending content!")   # Appends "Appending content!" to the file

```

5. Closing a File:

After you are done working with a file, it's important to close it using the `close()` method to release system resources. Here's an example:

```python

file.close()  # Closes the file

```

It's also recommended to use the `with` statement, which automatically handles closing the file for you. Here's an example:

```python

with open("file.txt", "r") as file:

    content = file.read()  # The file will be automatically closed after this block

```

These are the basic file operations in Python. Additionally, Python provides many other functionalities for file handling, such as renaming files, deleting files, working with directories, etc. You can explore the `os` and `shutil` modules for more advanced file operations.

Reading text files

To read the contents of a text file in Python, you can use the following steps:

1. Open the File:

Start by opening the file using the `open()` function. Provide the file path and specify the mode as "r" for reading.

2. Read the Contents:

There are different methods available to read the contents of the file, as discussed earlier. You can choose the appropriate method based on your requirements:

- Reading the Entire File:

- Reading Line by Line:

- Reading All Lines as a List:

3. Close the File:

After you have finished reading the file, it's good practice to close it using the `close()` method. However, when using the `with` statement, as shown in the examples above, the file will be automatically closed after exiting the block.

By following these steps, you can successfully read the contents of a text file in Python.

Read functions

Certainly! In Python, there are several read functions that you can use to read the contents of a file. Here are the commonly used ones:

1. `read()`:

The `read()` function is used to read the entire content of a file as a single string. It reads from the current position to the end of the file. Here's an example:

2. `read(size)`:

The `read(size)` function reads a specified number of characters from the file, starting from the current position. If no size is provided, it reads the entire file. Here's an example:

3. `readline()`:

The `readline()` function reads a single line from the file, including the newline character at the end of the line. On subsequent calls, it reads the next line. Here's an example:

4. `readlines()`:

The `readlines()` function reads all the lines from the file and returns them as a list of strings. Each element of the list represents a line from the file, including the newline characters. Here's an example:

These read functions allow you to retrieve the contents of a file in different formats, such as a single string, individual lines, or a list of lines. Choose the appropriate function based on your specific requirements. Remember to open the file in "read" mode (`"r"`) before using these read functions.

there are three primary methods for reading file content: `read()`, `readline()`, and `readlines()`.

1. `read()`:

The `read()` method reads the entire contents of a file as a single string. It returns a string that includes all the characters in the file, including newlines and any other special characters. Here's an example:

```python
with open("file.txt", "r") as file:
    content = file.read()
    print(content)
```

2. `readline()`:

The `readline()` method reads a single line from the file. It returns a string containing the characters of the current line, including the newline character. On subsequent calls, it reads the next line. Here's an example:

```python
with open("file.txt", "r") as file:
    line1 = file.readline()
    line2 = file.readline()
    print(line1)
    print(line2)
```

3. `readlines()`:

The `readlines()` method reads all the lines from the file and returns them as a list of strings. Each element of the list represents a line from the file, including the newline characters. Here's an example:

```python
with open("file.txt", "r") as file:
    lines = file.readlines()
    for line in lines:
```

```
    print(line)
```

It's worth noting that all these methods maintain the current position within the file. So, if you call `readline()` multiple times, it will continue reading subsequent lines. Similarly, if you call `read()` or `readlines()` after `readline()`, it will start reading from the current position.

Remember to open the file in "read" mode (`"r"`) before using these methods. Additionally, the `with` statement is used to automatically close the file after reading, ensuring proper resource management.

writing Text Files

To write text files in Python, you can follow these steps:

1. Open the File:

Start by opening the file using the `open()` function with the appropriate mode. Use "w" for writing (if the file already exists, its contents will be truncated) or "a" for appending (if the file exists, new content will be added to the end). For example:

```python
with open("file.txt", "w") as file:
    # File operations
```

2. Write Content:

To write content to the file, you can use the `write()` method. It takes a string as an argument and writes it to the file. Here's an example:

```python
with open("file.txt", "w") as file:
    file.write("Hello, world!")
```

If you want to write multiple lines, you can use the newline character ("\n") to separate the lines:

```python

with open("file.txt", "w") as file:

    file.write("Line 1\n")

    file.write("Line 2\n")

    file.write("Line 3\n")

```

3. Close the File:

After you have finished writing to the file, it's important to close it using the `close()` method. However, when using the `with` statement, as shown in the examples above, the file will be automatically closed after exiting the block.

```python

with open("file.txt", "w") as file:

    file.write("Hello, world!")

    # File operations

# File is automatically closed outside the 'with' block

```

By following these steps, you can successfully write content to a text file in Python. If you need to append content to an existing file, use mode "a" instead of "w" when opening the file.

write functions

When working with text files in Python, there are various write functions you can use to write content to the file. Here are the commonly used ones:

1. `write()`:

The `write()` function is used to write a string to a file. It takes a string as an argument and writes it to the file at the current position. If the file doesn't exist, it will be created. If it already exists, the previous contents will be overwritten. Here's an example:

```python

with open("file.txt", "w") as file:
```

```
    file.write("Hello, world!")
```

2. `writelines()`:

The `writelines()` function is used to write multiple strings to a file. It takes a list of strings as an argument and writes each string as a separate line in the file. Here's an example:

```python
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

with open("file.txt", "w") as file:
    file.writelines(lines)
```

It's worth noting that the strings passed to `write()` and `writelines()` functions should include the newline character ("\n") if you want to write separate lines in the file.

These write functions allow you to write content to a text file in Python. Remember to open the file in "write" mode (`"w"`) before using these write functions. If you want to append content to an existing file instead of overwriting it, use mode `"a"` when opening the file.

Manipulating file pointer using seek

In Python, you can manipulate the file pointer position using the `seek()` function. The `seek()` function allows you to change the position within a file where subsequent read or write operations will occur. It takes two arguments: the offset and the optional whence.

The `offset` parameter specifies the number of bytes to move the file pointer. A positive offset moves the pointer forward, while a negative offset moves it backward.

The `whence` parameter specifies the reference position from where the offset is applied. It can take one of three values:

- 0: The offset is relative to the beginning of the file (default).

- 1: The offset is relative to the current position of the file pointer.

- 2: The offset is relative to the end of the file.

Here's an example that demonstrates how to use `seek()` to manipulate the file pointer:

```python
with open("file.txt", "r") as file:
    # Move the file pointer to the 10th byte from the beginning of the file
    file.seek(10, 0)
    content = file.read()
    print(content)
    # Move the file pointer 5 bytes forward from the current position
    file.seek(5, 1)
    content = file.read()
    print(content)
    # Move the file pointer 10 bytes back from the end of the file
    file.seek(-10, 2)
    content = file.read()
    print(content)
```

In the first `seek()` call, the file pointer is moved to the 10th byte from the beginning (`whence=0`). In the second `seek()` call, it is moved 5 bytes forward from the current position (`whence=1`). In the third `seek()` call, it is moved 10 bytes back from the end of the file (`whence=2`).

Manipulating the file pointer with `seek()` allows you to read or write data from specific positions within a file, giving you flexibility and control over file operations.

Appending to Files

Appending data to files in Python is accomplished by opening the file in append mode ("a") rather than write mode ("w"). When a file is opened in append mode, new data is appended to the end of the existing content, without overwriting it. Here's how you can append data to a file:

```python
with open("file.txt", "a") as file:
    file.write("New content to be appended\n")
```

In the example above, the file "file.txt" is opened in append mode. The `write()` function is then used to append the string "New content to be appended\n" to the file. The newline character ("\n") is included to ensure that the new content starts on a new line.

You can also append multiple lines by using the `writelines()` function and passing a list of strings:

```python
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("file.txt", "a") as file:
    file.writelines(lines)
```

In this case, the list `lines` contains multiple strings, each representing a line of content to be appended. The `writelines()` function writes each string as a separate line in the file.

Remember to open the file in append mode ("a") to avoid overwriting existing content.

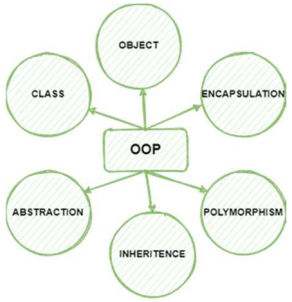
Python Object Oriented:

Overview of OOP

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction



OOPs

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
    # Statement-1
    .
    .
    # Statement-N
```

Python Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

- Python3

```
obj = Dog()
```

The Python self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.

- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

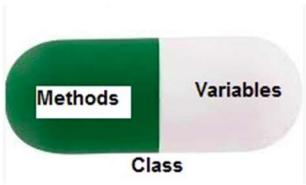
Python Polymorphism

Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

Python Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



Data Abstraction

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by creating abstract classes.

Accessing attributes

In Python, you can access attributes of an object using dot notation (`object.attribute`). When you create an instance of a class, you can access its attributes and modify them as needed. Let's consider an example:

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

person = Person("Alice", 25)

# Accessing attributes
print(person.name)  # Output: Alice
print(person.age)   # Output: 25

# Modifying attributes
person.name = "Bob"
person.age = 30

print(person.name)  # Output: Bob
print(person.age)   # Output: 30
```

In the example above, the `Person` class has two attributes (`name` and `age`). When we create an instance of the `Person` class (`person`), we can access its attributes using dot notation (`person.name`, `person.age`).

To modify the attribute values, you can assign new values to them using the same dot notation (`person.name = "Bob"`, `person.age = 30`).

You can also access attributes within methods of the class using the `self` parameter, which refers to the instance of the class. For example:

```python
class Person:
```

```
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def print_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
person = Person("Alice", 25)
person.print_info()
```

In this example, the `print_info` method accesses the `name` and `age` attributes of the `Person` instance using `self.name` and `self.age`, respectively.

Built-In Class Attributes

In Python, classes have some built-in class attributes that provide information about the class itself. These attributes can be accessed using dot notation on the class itself. Here are some commonly used built-in class attributes:

1. `__name__`: It returns the name of the class as a string.

2. `__module__`: It returns the name of the module where the class is defined.

3. `__doc__`: It returns the docstring (documentation string) of the class.

4. `__dict__`: It returns a dictionary containing the class's namespace.

5. `__bases__`: It returns a tuple containing the base classes of the class.

Here's an example demonstrating the use of these built-in class attributes:

```python
class MyClass:
    """This is a sample class."""
    attr = "sample attribute"
```

```
obj = MyClass()

print(MyClass.__name__)       # Output: MyClass

print(MyClass.__module__)     # Output: __main__

print(MyClass.__doc__)        # Output: This is a sample class.

print(MyClass.__dict__)       # Output: {'__module__': '__main__', '__doc__': 'This is a sample class.', 'attr': 'sample attribute', ...}

print(MyClass.__bases__)      # Output: (<class 'object'>,)
```

In the above example, we define a class `MyClass` with an attribute `attr`. We then create an instance `obj` of the class.

By accessing the built-in class attributes using dot notation (`MyClass.__name__`, `MyClass.__module__`, etc.), we can retrieve information about the class itself. For example, `MyClass.__name__` returns the name of the class, `MyClass.__doc__` returns the docstring, and `MyClass.__dict__` returns a dictionary containing the class's namespace, which includes its attributes and methods.

Note that the `__bases__` attribute returns a tuple containing the base classes of the class. In the example above, since `MyClass` doesn't explicitly inherit from any other class, it has `object` as its base class.

Methods

In Python, methods are functions defined within a class that operate on objects (instances) of that class. They define the behavior or actions that objects of the class can perform. Methods can access and modify the attributes of the class and perform various operations. There are three types of methods commonly used in Python classes:

1. Instance Methods: Instance methods are defined within a class and are bound to the instances (objects) of the class. They have access to the instance attributes and can modify them. The first parameter of an instance method is usually `self`, which refers to the instance itself.

```python
class MyClass:

    def instance_method(self, arg1, arg2):

        # Accessing instance attributes

        self.attribute = arg1

        # Performing operations
```

```
        result = arg1 + arg2

        return result

obj = MyClass()

result = obj.instance_method(3, 4)

print(result)  # Output: 7
```

In the above example, `instance_method` is an instance method defined within the `MyClass` class. It takes two arguments (`arg1` and `arg2`) and performs an addition operation. The `self.attribute` line demonstrates how to access and modify instance attributes within an instance method.

2. Class Methods: Class methods are bound to the class itself rather than instances of the class. They can access and modify class-level attributes but not instance attributes. Class methods are defined using the `@classmethod` decorator, and the first parameter is usually named `cls`, referring to the class itself.

```python
class MyClass:
    class_attribute = 0

    @classmethod

    def class_method(cls, arg):

        # Accessing class attributes

        cls.class_attribute = arg

        # Performing operations

        result = arg * 2

        return result

result = MyClass.class_method(5)

print(result)              # Output: 10

print(MyClass.class_attribute)  # Output: 5
```

In the above example, `class_method` is a class method defined within the `MyClass` class. It takes an argument (`arg`) and performs a multiplication operation. The `cls.class_attribute` line demonstrates how to access and modify class attributes within a class method.

3. Static Methods: Static methods are not bound to the instance or class and do not have access to instance or class attributes. They behave like regular functions but are defined within the class for convenience. Static methods are defined using the `@staticmethod` decorator.

```python

class MyClass:

    @staticmethod

    def static_method(arg1, arg2):

        # Performing operations

        result = arg1 + arg2

        return result

result = MyClass.static_method(2, 3)

print(result)  # Output: 5

```

In the above example, `static_method` is a static method defined within the `MyClass` class. It takes two arguments (`arg1` and `arg2`) and performs an addition operation. Since static methods are not bound to instances or classes, they don't have access to any attributes.

These are the three main types of methods in Python classes: instance methods, class methods, and static methods. Each type has its own use cases and provides different functionality within a class.

Class and Instance Variables

In Python classes, you can have both class variables and instance variables. Class variables are shared among all instances of a class, while instance variables are specific to each instance of the class.

Class Variables:

- Class variables are defined within the class but outside of any methods.

- They are shared by all instances of the class.

- Class variables are accessed using the class name or instance name followed by dot notation.

- They are typically used to store data that is common to all instances of the class.

Here's an example:

```python
class Circle:
    pi = 3.14  # Class variable
    def __init__(self, radius):
        self.radius = radius  # Instance variable
circle1 = Circle(5)
circle2 = Circle(3)
print(circle1.radius)  # Output: 5
print(circle2.radius)  # Output: 3
print(circle1.pi)      # Output: 3.14
print(circle2.pi)      # Output: 3.14
print(Circle.pi)       # Output: 3.14
```

In the above example, `pi` is a class variable defined within the `Circle` class. It is accessed using `Circle.pi`, `circle1.pi`, or `circle2.pi`. All instances of the `Circle` class share the same `pi` value.

Instance Variables:

- Instance variables are specific to each instance of a class.
- They are defined within the class's methods, particularly in the `__init__` method.
- Instance variables can store unique data for each instance.
- They are accessed using the instance name followed by dot notation.

It's important to note that modifying a class variable affects all instances of the class, while modifying an instance variable only affects the specific instance it belongs to. Understanding the distinction between class and instance variables is crucial when designing and working with classes in Python.

Destroying Objects

In Python, objects are automatically destroyed (reclaimed by the memory management system) when they are no longer referenced. Python uses a garbage collector that keeps track of objects and frees up memory when objects are no longer needed. However, you can also explicitly destroy objects using the `del` statement.

Here's an example that demonstrates the destruction of objects:

```python
class MyClass:
    def __init__(self, name):
        self.name = name
        print(f"Creating object {self.name}")
    def __del__(self):
        print(f"Destroying object {self.name}")
# Creating objects
obj1 = MyClass("Object 1")
obj2 = MyClass("Object 2")
# Deleting objects explicitly
del obj1
del obj2
# Output:
# Creating object Object 1
# Creating object Object 2
# Destroying object Object 1
# Destroying object Object 2
```

In the above example, the `MyClass` class has an `__init__` method that is called when objects are created, and an `__del__` method that is called when objects are destroyed.

We create two objects `obj1` and `obj2` of the `MyClass` class. After we delete the objects using `del obj1` and `del obj2`, the `__del__` method is automatically called for each object, indicating their destruction.

It's important to note that the garbage collector in Python automatically takes care of memory management, so you don't usually need to explicitly destroy objects. The `__del__` method is not guaranteed to be called immediately when an object is no longer referenced, as it depends on the garbage collector's behavior. The purpose of the `__del__` method is typically for performing any necessary cleanup or finalization tasks associated with the object.

It's generally recommended to rely on the automatic garbage collection and let the objects be destroyed naturally when they are no longer needed.

Polymorphism

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables objects of different types to be used interchangeably in a program, providing flexibility and code reusability.

Polymorphism can be achieved through method overriding and method overloading.

1. Method Overriding:

Method overriding occurs when a subclass defines a method with the same name as a method in its superclass. The subclass method overrides the implementation of the superclass method. When a method is called on an object of the subclass, the overridden method in the subclass is executed.

```python

class Shape:

    def draw(self):

        print("Drawing a shape.")

class Circle(Shape):

    def draw(self):

        print("Drawing a circle.")

class Square(Shape):

    def draw(self):

        print("Drawing a square.")

# Polymorphic behavior

shapes = [Circle(), Square()]
```

```
for shape in shapes:
    shape.draw()
```

In the above example, the `Shape` class has a method called `draw()`, which is overridden in the `Circle` and `Square` subclasses. We create a list `shapes` containing instances of both `Circle` and `Square`. By iterating over the list and calling the `draw()` method, polymorphism is achieved. The appropriate `draw()` method of each shape is invoked based on its actual type.

2. Method Overloading:

Method overloading refers to defining multiple methods with the same name but different parameters within a class. Python does not support method overloading in the same way as some other languages like Java or C++, where different methods can have the same name but different parameter lists. However, you can achieve a similar effect by using default parameter values or variable-length arguments.

```python
class Calculator:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c
# Overloading using default parameter
calculator = Calculator()
result1 = calculator.add(2, 3)       # Output: 5
result2 = calculator.add(2, 3, 4)    # Output: 9
# Overloading using variable-length arguments
class Printer:
    def print_values(self, *args):
        for arg in args:
```

```
        print(arg)

printer = Printer()

printer.print_values(1)            # Output: 1

printer.print_values(1, 2, 3)      # Output: 1 2 3
```

In the above example, the `Calculator` class demonstrates method overloading using default parameters. The `add()` method is defined twice, once with two parameters and once with three parameters. Depending on the number of arguments provided when calling the method, the appropriate version of `add()` is executed.

The `Printer` class demonstrates method overloading using variable-length arguments. The `print_values()` method accepts any number of arguments and prints them.

In summary, polymorphism allows different objects to be treated interchangeably based on their shared superclass or interface. Method overriding and method overloading are techniques used to achieve polymorphic behavior in Python.

Overlapping and Overloading of Operators

In Python, you can define the behavior of operators for your custom classes by implementing special methods called magic methods or dunder methods. Overloading operators allows objects of your class to use operators such as addition, subtraction, equality, etc., in a way that makes sense for your specific class.

Here are a few examples of operator overloading:

1. Overloading the Addition Operator (+):

To overload the addition operator for a class, you can define the `__add__` method. This method takes two objects as parameters and returns the result of the addition.

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

```
    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type.")
p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2
print(p3.x, p3.y)  # Output: 6, 8
```

In the above example, the `__add__` method is defined to handle the addition of two `Point` objects. It creates a new `Point` object with the coordinates obtained by adding the corresponding coordinates of the two points.

2. Overloading the Equality Operator (==):

To overload the equality operator for a class, you can define the `__eq__` method. This method takes two objects as parameters and returns `True` if they are equal, and `False` otherwise.

```python
class Person:
    def __init__(self, name):
        self.name = name
    def __eq__(self, other):
        if isinstance(other, Person):
            return self.name == other.name
        else:
            return False
person1 = Person("Alice")
```

```
person2 = Person("Bob")

person3 = Person("Alice")

print(person1 == person2)  # Output: False

print(person1 == person3)  # Output: True
```

In the above example, the `__eq__` method is defined to compare the names of two `Person` objects and return `True` if they are equal, and `False` otherwise.

These are just a few examples of operator overloading in Python. There are many other dunder methods you can implement to overload various operators such as subtraction (-), multiplication (*), less than (<), etc. By implementing these methods, you can define the behavior of operators for your custom classes, allowing you to use operators on objects in a way that is meaningful for your class.

Class Inheritance: super()

In Python, class inheritance allows you to create a new class that inherits attributes and methods from an existing class. The new class is called a subclass or derived class, and the existing class is called the superclass or base class. The subclass can add new attributes and methods or override the inherited ones.

The `super()` function is used in Python to refer to the superclass and access its methods and attributes from within the subclass. It allows you to call and invoke the superclass's methods while overriding or extending their functionality in the subclass.

Here's an example to illustrate the usage of `super()` in class inheritance:

```python
class Vehicle:
    def __init__(self, color):
        self.color = color
    def drive(self):
        print("Driving the vehicle.")
class Car(Vehicle):
    def __init__(self, color, brand):
        super().__init__(color)  # Calling superclass's __init__()
```

```
        self.brand = brand

    def drive(self):

        super().drive()  # Calling superclass's drive()

        print(f"Driving the {self.brand} car.")

car = Car("Red", "Toyota")

print(car.color)  # Output: Red

car.drive()
```

In the above example, we have a `Vehicle` class with an `__init__()` method and a `drive()` method. The `Car` class is a subclass of `Vehicle` and inherits these methods. The `Car` class overrides the `drive()` method to add additional functionality.

In the `Car` class, the `__init__()` method calls the superclass's `__init__()` method using `super().__init__(color)`. This ensures that the superclass's initialization is performed before adding any specific initialization for the `Car` class.

Similarly, the `drive()` method in the `Car` class calls the superclass's `drive()` method using `super().drive()`. This allows the `Car` class to invoke the superclass's behavior while adding its own behavior.

When creating an instance of the `Car` class and calling its methods, the output will be as follows:

```
Red

Driving the vehicle.

Driving the Toyota car.
```

As shown in the example, `super()` provides a convenient way to access the superclass's methods and attributes within the subclass. It helps in achieving code reuse and maintaining a clear and organized inheritance hierarchy.

Method Overriding

Method overriding is a feature in object-oriented programming that allows a subclass to provide a different implementation of a method that is already defined in its superclass. The overridden method in the subclass replaces the implementation inherited from the superclass.

When a method is called on an instance of the subclass, if the subclass has overridden that method, the overridden version is executed instead of the superclass's version. This allows the subclass to customize or extend the behavior of the inherited method.

Here's an example that demonstrates method overriding:

```python
class Animal:
    def speak(self):
        print("The animal makes a sound.")

class Dog(Animal):
    def speak(self):
        print("The dog barks.")

class Cat(Animal):
    def speak(self):
        print("The cat meows.")

# Creating instances
animal = Animal()
dog = Dog()
cat = Cat()

# Method overriding
animal.speak()  # Output: The animal makes a sound.
dog.speak()    # Output: The dog barks.
cat.speak()    # Output: The cat meows.
```

In the above example, we have a base class `Animal` with a method called `speak()`, which prints a generic sound. The `Dog` and `Cat` classes are subclasses of `Animal` and override the `speak()` method with their own specific implementations.

When the `speak()` method is called on instances of `Animal`, `Dog`, and `Cat`, the appropriate overridden version of the method is executed based on the actual type of the object.

Method overriding allows you to tailor the behavior of methods to the specific needs of each subclass, while still maintaining a common interface through inheritance. It promotes code reusability and flexibility in object-oriented design.

Exception Handling

Exception handling in Python allows you to handle and recover from errors or exceptional situations that may occur during the execution of your program. It helps you write robust code by gracefully dealing with potential errors and preventing your program from crashing.

Python provides a try-except block for handling exceptions. The code that may raise an exception is enclosed within the try block, and the exception handling code is written in the except block.

Here's the basic syntax of a try-except block:

```python
try:
    # Code that may raise an exception
    # ...
except ExceptionType1:
    # Exception handling code for ExceptionType1
    # ...
except ExceptionType2:
    # Exception handling code for ExceptionType2
    # ...
```

```
else:
    # Code that executes if no exception occurs
    # ...
finally:
    # Code that always executes, regardless of exceptions
    # ...
```

In the try block, you write the code that you think might raise an exception. If an exception occurs within the try block, the code execution is immediately transferred to the appropriate except block based on the type of the exception.

You can have multiple except blocks to handle different types of exceptions. Each except block specifies the type of exception it can handle. If an exception matches the type specified in an except block, the corresponding exception handling code within that block is executed.

The else block is optional and contains code that executes if no exception occurs in the try block.

The finally block is also optional and contains code that always executes, regardless of whether an exception occurred or not. It is typically used for cleanup operations, such as closing files or releasing resources.

Here's an example to illustrate exception handling in Python:

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

```
    print("No exceptions occurred.")

finally:

    print("Exception handling complete.")
```

In the above example, the user is prompted to enter a number. If the user enters a valid number, it is divided by 10, and the result is printed. If the user enters an invalid value (e.g., a non-numeric value) or if the number entered is zero, the appropriate exception handling code is executed.

Whether an exception occurs or not, the code within the finally block always executes at the end.

Exception handling allows you to gracefully handle errors and control the flow of your program in the face of exceptional situations. It is recommended to handle exceptions specific to the anticipated errors and provide meaningful error messages to users for better error handling and debugging.

Try-except-else clause

The try-except-else clause is a construct in Python that allows you to handle exceptions in a more precise and controlled manner. It provides a way to specify code that should be executed only if no exception occurs within the try block.

The basic structure of the try-except-else clause is as follows:

```python
try:

    # Code that may raise an exception

    # ...

except ExceptionType1:

    # Exception handling code for ExceptionType1

    # ...

except ExceptionType2:

    # Exception handling code for ExceptionType2

    # ...
```

```
else:
    # Code that executes if no exception occurs
    # ...
```

Here's how the try-except-else clause works:

1. The code that may raise an exception is placed within the try block.

2. If an exception occurs within the try block, the code execution jumps to the appropriate except block based on the type of the exception.

3. If no exception occurs within the try block, the code within the else block is executed. This code is meant to handle the normal execution flow when no exceptions are encountered.

4. If an exception occurs and is handled by one of the except blocks, the code within the else block is skipped.

The try-except-else clause provides a way to separate the exception handling code from the normal code flow. It allows you to handle exceptions specifically and provide different exception handling strategies based on the type of the exception.

Here's an example to illustrate the usage of the try-except-else clause:

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

```
    print("Result:", result)

    print("No exceptions occurred.")
```

In the above example, the user is prompted to enter a number. If the user enters a valid number, it is divided by 10, and the result is printed. If the user enters an invalid value (e.g., a non-numeric value) or if the number entered is zero, the corresponding exception handling code is executed. If no exception occurs, the result is printed, along with a message indicating that no exceptions occurred.

The try-except-else clause allows you to handle exceptions in a fine-grained manner and separate the exceptional cases from the normal code flow. It provides more control and flexibility in handling exceptions and helps in writing robust and error-tolerant code.

Python Standard Exceptions

Python provides a set of standard exceptions that are predefined in the language. These exceptions represent common error conditions or exceptional situations that can occur during the execution of a Python program. Understanding these standard exceptions can help you handle errors effectively and provide appropriate error messages to users.

Here are some of the most commonly used standard exceptions in Python:

1. `Exception`: The base class for all exceptions in Python. It can be used to catch any exception.

2. `TypeError`: Raised when an operation or function is applied to an object of inappropriate type.

3. `ValueError`: Raised when a function receives an argument of correct type but an invalid value.

4. `IndexError`: Raised when a sequence subscript is out of range.

5. `KeyError`: Raised when a dictionary key is not found.

6. `FileNotFoundError`: Raised when a file or directory is requested but cannot be found.

7. `ZeroDivisionError`: Raised when division or modulo operation is performed with zero as the divisor.

8. `AttributeError`: Raised when an attribute reference or assignment fails.

9. `ImportError`: Raised when an imported module, function, or class is not found.

10. `NameError`: Raised when a local or global name is not found.

11. `IOError`: Raised when an I/O operation fails.

12. `RuntimeError`: Raised when an error occurs that doesn't belong to any specific category.

These are just a few examples of the standard exceptions available in Python. There are many more exceptions provided by the language, each serving a specific purpose.

When handling exceptions, it is generally recommended to catch specific exceptions rather than catching the base `Exception` class. This allows for more precise error handling and better control over the program's behavior.

Here's an example that demonstrates catching specific exceptions:

```python
try:
    # Code that may raise an exception
    # ...
except ValueError:
    # Exception handling code for ValueError
    # ...
except IndexError:
    # Exception handling code for IndexError
    # ...
except FileNotFoundError:
    # Exception handling code for FileNotFoundError
    # ...
```

By understanding the standard exceptions provided by Python, you can effectively handle errors and exceptional situations in your code and provide appropriate feedback to users when errors occur.

User-Defined Exceptions

In addition to the standard exceptions provided by Python, you can also define your own custom exceptions to represent specific error conditions or exceptional situations that are relevant to your application. User-defined exceptions allow you to create a hierarchy of exception classes tailored to your program's needs.

To define a user-defined exception, you can create a new class that inherits from the base `Exception` class or any other existing exception class. Typically, you would create a new exception class to represent a specific error condition or a category of related errors.

Here's an example that demonstrates the creation of a user-defined exception class:

```python
class CustomException(Exception):
    pass

# Raise the custom exception
raise CustomException("This is a custom exception.")
```

In the above example, we define a new exception class `CustomException` that inherits from the base `Exception` class. The `CustomException` class doesn't add any new functionality, as it simply inherits the behavior and attributes from its base class.

To raise the custom exception, we use the `raise` statement followed by an instance of the `CustomException` class. In this case, we pass a string argument that provides additional information about the exception.

You can also add custom attributes and methods to your user-defined exception class to provide more context and functionality specific to your application.

Here's an example that demonstrates a user-defined exception with custom attributes and methods:

```python
class CustomException(Exception):
    def __init__(self, message, code):
```

```
        super().__init__(message)
        self.code = code
    def get_error_code(self):
        return self.code

try:
    raise CustomException("Custom exception occurred.", 500)
except CustomException as e:
    print("Error message:", str(e))
    print("Error code:", e.get_error_code())
```

In this example, the `CustomException` class has a constructor that takes a message and a code as arguments. It calls the constructor of its superclass (`Exception`) to initialize the message. It also defines a method `get_error_code()` to retrieve the code attribute of the exception.

When raising the custom exception, we pass the message and code values. In the except block, we catch the `CustomException` and access its attributes and methods to retrieve and display the error information.

By defining and using user-defined exceptions, you can create a more structured and meaningful exception handling mechanism in your code. This allows you to handle specific error conditions or exceptional situations in a more precise and controlled manner.

UNIT 4

Databases in Python:

Python provides several libraries and frameworks for working with databases. Here are some popular options:

1. **SQLite**: SQLite is a lightweight and serverless database engine that is included with Python. It doesn't require any separate installation and uses a single file for the entire database. The `sqlite3` module in the Python standard library provides an interface to interact with SQLite databases.

2. **MySQL**: MySQL is a widely used open-source relational database management system. To work with MySQL databases in Python, you can use the `mysql-connector-python` library or the `pymysql` library. These libraries provide APIs to establish connections, execute queries, and manage transactions with MySQL.

3. **PostgreSQL**: PostgreSQL is another popular open-source relational database management system. To interact with PostgreSQL databases in Python, you can use the `psycopg2` library. It offers a comprehensive set of functions for database connectivity and query execution.

4. **MongoDB**: MongoDB is a NoSQL database that stores data in a flexible, JSON-like format. The `pymongo` library is the official MongoDB driver for Python. It allows you to connect to MongoDB, insert and retrieve data, and perform various operations on the database.

5. **SQLAlchemy**: SQLAlchemy is a powerful SQL toolkit and Object-Relational Mapping (ORM) library. It supports multiple database engines, including SQLite, MySQL, PostgreSQL, and more. SQLAlchemy provides a high-level, Pythonic interface for working with databases, allowing you to write database-agnostic code.

These are just a few examples of the available options for working with databases in Python. The choice of database and library depends on your specific requirements and preferences.

Create Database Connection

To create a database connection in Python, the specific steps and code will vary depending on the database you are using. Here are examples of creating database connections for SQLite, MySQL, and PostgreSQL databases using their respective Python libraries.

SQLite:

```python

import sqlite3
```

```
# Create a connection to an SQLite database
conn = sqlite3.connect('database.db')
# Create a cursor object to execute SQL statements
cursor = conn.cursor()
# Example: Execute a SQL query
cursor.execute('SELECT * FROM table_name')
# Fetch and print the query results
results = cursor.fetchall()
for row in results:
    print(row)
# Close the cursor and the database connection
cursor.close()
conn.close()
```

MySQL:

```python
import mysql.connector
# Create a connection to a MySQL database
conn = mysql.connector.connect(
    host="localhost",
    user="username",
    password="password",
    database="database_name"
)
```

```
# Create a cursor object to execute SQL statements

cursor = conn.cursor()


# Example: Execute a SQL query

cursor.execute('SELECT * FROM table_name')


# Fetch and print the query results

results = cursor.fetchall()

for row in results:

    print(row)


# Close the cursor and the database connection

cursor.close()

conn.close()
```

Make sure to replace `database.db`, `username`, `password`, and `database_name` with the appropriate values for your database.

create, insert, read, update and delete Operation

Certainly! Here's an example that demonstrates how to perform create, insert, read, update, and delete operations using the `sqlite3` module in Python with an SQLite database.

```python

import sqlite3

# Create a connection to an SQLite database

conn = sqlite3.connect('database.db')

cursor = conn.cursor()
```

```
# Create a table
create_table_query = '''
   CREATE TABLE IF NOT EXISTS employees (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      name TEXT NOT NULL,
      age INTEGER,
      department TEXT
   )
'''
cursor.execute(create_table_query)

# Insert data into the table
insert_query = '''
   INSERT INTO employees (name, age, department) VALUES (?, ?, ?)
'''
cursor.execute(insert_query, ('John Doe', 30, 'IT'))
cursor.execute(insert_query, ('Jane Smith', 35, 'HR'))
conn.commit()

# Read data from the table
select_query = '''
   SELECT * FROM employees
'''
cursor.execute(select_query)
```

```
rows = cursor.fetchall()
for row in rows:
    print(row)

# Update a record
update_query = '''
    UPDATE employees SET department = ? WHERE id = ?
'''
cursor.execute(update_query, ('Finance', 1))
conn.commit()

# Delete a record
delete_query = '''
    DELETE FROM employees WHERE id = ?
'''
cursor.execute(delete_query, (2,))
conn.commit()

# Read the updated data from the table
cursor.execute(select_query)
rows = cursor.fetchall()
for row in rows:
    print(row)
```

```
# Close the cursor and the database connection

cursor.close()

conn.close()
```

In this example, we create an `employees` table with columns `id`, `name`, `age`, and `department`. We perform operations such as creating the table if it doesn't exist, inserting data into the table, reading data from the table, updating a record, and deleting a record. Finally, we fetch and print the updated data from the table.

Remember to replace `'database.db'` with the appropriate path or name of your SQLite database file.

DML and DDL Operation with Databases.

In the context of databases, DML (Data Manipulation Language) and DDL (Data Definition Language) are two categories of SQL (Structured Query Language) statements used to perform different operations. Here's a brief overview of DML and DDL operations:

****Data Manipulation Language (DML)**:**

DML statements are used to manipulate data stored in a database. The common DML statements are:

- **SELECT**: Used to retrieve data from one or more tables.

- **INSERT**: Used to insert new records into a table.

- **UPDATE**: Used to modify existing records in a table.

- **DELETE**: Used to delete records from a table.

- **MERGE**: Used to perform insert, update, or delete operations based on a condition.

Example DML statements:

```sql
-- SELECT statement

SELECT * FROM employees;
```

```
-- INSERT statement

INSERT INTO employees (name, age) VALUES ('John Doe', 30);


-- UPDATE statement

UPDATE employees SET age = 31 WHERE id = 1;


-- DELETE statement

DELETE FROM employees WHERE id = 2;


-- MERGE statement

MERGE INTO employees USING temp_employees

ON employees.id = temp_employees.id

WHEN MATCHED THEN UPDATE SET employees.name = temp_employees.name

WHEN NOT MATCHED THEN INSERT (id, name) VALUES (temp_employees.id, temp_employees.name);
```

Data Definition Language (DDL):**

DDL statements are used to define, modify, or delete database structures and schema objects. The common DDL statements are:

- **CREATE**: Used to create new database objects like tables, views, indexes, etc.

- **ALTER**: Used to modify the structure of existing database objects.

- **DROP**: Used to delete or remove database objects.

- **TRUNCATE**: Used to remove all data from a table, but keeps the structure intact.

Example DDL statements:

```sql
-- CREATE TABLE statement
CREATE TABLE employees (
   id INT PRIMARY KEY,
   name VARCHAR(50),
   age INT
);

-- ALTER TABLE statement
ALTER TABLE employees ADD COLUMN department VARCHAR(50);

-- DROP TABLE statement
DROP TABLE employees;

-- TRUNCATE TABLE statement
TRUNCATE TABLE employees;
```

These are just a few examples of DML and DDL statements. SQL provides a rich set of statements and functions to manipulate and define database data and structures. The specific syntax and usage may vary depending on the database management system you are using.

Python for Data Analysis:

Python is a popular programming language for data analysis due to its simplicity, flexibility, and a rich ecosystem of libraries and tools specifically designed for working with data. Here are some key libraries and concepts used in Python for data analysis:

1. **NumPy**: NumPy is a fundamental library for numerical computing in Python. It provides support for efficient operations on large multi-dimensional arrays and matrices. NumPy is the foundation for many other data analysis libraries in Python.

2. **Pandas**: Pandas is a powerful library built on top of NumPy that provides high-performance data structures and data analysis tools. It introduces the DataFrame, a tabular data structure, which allows for efficient manipulation and analysis of data. Pandas provides functions for data cleaning, transformation, filtering, and aggregation.

3. **Matplotlib**: Matplotlib is a popular plotting library in Python. It provides a wide range of functions for creating static, animated, and interactive visualizations. Matplotlib allows you to create various types of plots, including line plots, scatter plots, bar plots, histograms, and more.

4. **Seaborn**: Seaborn is a statistical data visualization library that is built on top of Matplotlib. It provides a higher-level interface for creating attractive and informative statistical graphics. Seaborn simplifies the process of creating complex visualizations by providing default styles and color palettes.

5. **Scikit-learn**: Scikit-learn is a machine learning library in Python that provides a wide range of algorithms for tasks such as classification, regression, clustering, and dimensionality reduction. It also offers utilities for data preprocessing, model evaluation, and model selection.

6. **Jupyter Notebook**: Jupyter Notebook is an interactive computing environment that allows you to create and share documents containing live code, equations, visualizations, and narrative text. It is widely used for data analysis and exploration as it enables you to combine code, data, and visualizations in a single document.

7. **Data manipulation and analysis**: Python provides a wide range of functions and libraries for data manipulation and analysis. Apart from Pandas, libraries such as SciPy, StatsModels, and Dask are commonly used for statistical analysis, hypothesis testing, and handling large datasets. Additionally, tools like SQLalchemy provide ways to work with databases directly in Python.

These are just a few of the tools and concepts used in Python for data analysis. The Python data analysis ecosystem is constantly evolving, and new libraries and techniques are regularly developed to enhance the capabilities of data analysis in Python.

Numpy:

NumPy (Numerical Python) is a fundamental library in Python for numerical computing. It provides efficient operations on large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to work with the data. NumPy serves as the foundation for many other libraries and tools used in scientific computing and data analysis. Here are some key features and functionalities of NumPy:

1. **Arrays**: NumPy introduces the `ndarray` (n-dimensional array) data structure, which allows for efficient storage and manipulation of homogeneous data. Arrays in NumPy can have any number of dimensions and support various data types.

2. **Vectorized Operations**: NumPy provides vectorized operations, which enable you to perform operations on entire arrays without writing explicit loops. This results in faster and more concise code. For example, you can add, subtract, multiply, divide, and apply mathematical functions to arrays element-wise.

3. **Mathematical Functions**: NumPy offers a comprehensive collection of mathematical functions for performing operations on arrays. These include functions for basic arithmetic, trigonometry, logarithms, exponents, statistics, linear algebra, and more.

4. **Broadcasting**: NumPy's broadcasting feature allows arrays with different shapes to be used in operations together, as long as their dimensions are compatible. This eliminates the need for explicitly expanding dimensions or duplicating data.

5. **Array Manipulation**: NumPy provides functions for reshaping, slicing, indexing, and concatenating arrays. You can modify the shape of an array, extract subsets of data, and combine arrays along different axes.

6. **Random Number Generation**: NumPy includes a random module that allows you to generate random numbers or arrays following various probability distributions.

7. **Integration with Other Libraries**: NumPy seamlessly integrates with other libraries in the scientific Python ecosystem. It is often used in conjunction with libraries such as Pandas, Matplotlib, and Scikit-learn for data manipulation, visualization, and machine learning tasks.

NumPy is a powerful library that enables efficient numerical computations and array manipulation in Python. It provides a solid foundation for working with data in scientific computing, data analysis, and machine learning tasks.

Creating arrays

In NumPy, you can create arrays using various methods. Here are some common ways to create arrays:

1. **Using the `np.array()` function**: You can create an array by passing a Python list or tuple to the `np.array()` function.

```python

import numpy as np

# Create a 1D array from a list

arr1 = np.array([1, 2, 3, 4, 5])
```

Create a 2D array from a nested list

arr2 = np.array([[1, 2, 3], [4, 5, 6]])

```

2. **Using NumPy functions**: NumPy provides several functions to create arrays with specific properties.

```python

import numpy as np

# Create an array of zeros

zeros_arr = np.zeros((3, 4))  # Creates a 3x4 array of zeros

# Create an array of ones

ones_arr = np.ones((2, 3))  # Creates a 2x3 array of ones

# Create an array with a specific value

value_arr = np.full((2, 2), 7)  # Creates a 2x2 array filled with 7

# Create a range of values

range_arr = np.arange(0, 10, 2)  # Creates an array with values [0, 2, 4, 6, 8]

# Create an array of random values

random_arr = np.random.rand(3, 3)  # Creates a 3x3 array of random values

```

3. **Using special functions**: NumPy provides additional functions to create arrays with specific patterns or properties.

```python

import numpy as np

# Create an identity matrix

identity_arr = np.eye(3)  # Creates a 3x3 identity matrix

```
Create a diagonal matrix

diagonal_arr = np.diag([1, 2, 3]) # Creates a 3x3 diagonal matrix with values [1, 2, 3]

Create a sequence of evenly spaced values

sequence_arr = np.linspace(0, 1, 5) # Creates an array with 5 values evenly spaced from 0 to 1
```

These are just a few examples of how to create arrays using NumPy. The specific method you choose depends on your desired array shape, content, and properties. NumPy provides a wide range of options to create and initialize arrays to suit your needs.

Using arrays and Scalars

In NumPy, you can perform arithmetic operations between arrays and scalars (single values). The operations are automatically applied element-wise, which means the scalar value is applied to each element of the array. Here are examples of using arrays and scalars in arithmetic operations:

```python
import numpy as np

# Create an array

arr = np.array([1, 2, 3, 4, 5])

# Scalar addition

result1 = arr + 2  # Add 2 to each element of the array

# Output: array([3, 4, 5, 6, 7])

# Scalar subtraction

result2 = arr - 1  # Subtract 1 from each element of the array

# Output: array([0, 1, 2, 3, 4])

# Scalar multiplication

result3 = arr * 3  # Multiply each element of the array by 3

# Output: array([3, 6, 9, 12, 15])
```

```
# Scalar division

result4 = arr / 2  # Divide each element of the array by 2

# Output: array([0.5, 1. , 1.5, 2. , 2.5])

# Scalar exponentiation

result5 = arr ** 2  # Square each element of the array

# Output: array([ 1,  4,  9, 16, 25])
```

In the above examples, the scalar values (2, 1, 3, 2, and 2) are applied element-wise to the corresponding elements of the array. The result is a new array with the same shape as the original array, containing the computed values.

These operations can also be performed using other scalar values or arrays of compatible shapes. NumPy's broadcasting rules allow for element-wise operations between arrays with different shapes, as long as the shapes are compatible.

It's important to note that NumPy operations are usually more efficient than using traditional loops in Python, especially when dealing with large arrays, thanks to its vectorized operations.

Indexing Arrays

In NumPy, you can access and manipulate individual elements or subsets of elements in arrays using indexing. Indexing in NumPy is similar to indexing in Python lists, but with some additional capabilities for multidimensional arrays. Here are some common indexing techniques:

1. **Single Element Access**: You can access a single element in a 1D array by specifying its index within square brackets.

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Accessing a single element

element = arr[2]  # Accesses the element at index 2

# Output: 3
```

2. **Slicing**: You can extract a subset of elements from an array using slicing. Slicing is done by specifying the start and end indices, separated by a colon, within square brackets.

```python

import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Slicing a portion of the array

subset = arr[1:4]  # Extracts elements from index 1 to index 3

# Output: array([2, 3, 4])

```

3. **Multidimensional Indexing**: For multidimensional arrays, you can access elements using comma-separated indices or slices within square brackets.

```python

import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing individual elements

element = arr[1, 2]  # Accesses the element at row index 1 and column index 2

# Output: 6

# Accessing subsets of elements

row_subset = arr[0:2, :]  # Extracts rows 0 and 1

# Output: array([[1, 2, 3], [4, 5, 6]])


column_subset = arr[:, 1:3]  # Extracts columns 1 and 2

# Output: array([[2, 3], [5, 6], [8, 9]])

```

4. **Boolean Indexing**: You can use boolean arrays to index an array, where each element of the boolean array specifies whether to include the corresponding element in the resulting subset.

```python

import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Boolean indexing

mask = arr > 2  # Creates a boolean array with True for elements greater than 2

subset = arr[mask]  # Extracts elements where the mask is True

# Output: array([3, 4, 5])

```

NumPy also provides advanced indexing techniques like integer array indexing and conditional indexing, allowing you to perform complex indexing operations

Remember that indexing in NumPy starts from 0, similar to Python lists. Additionally, NumPy arrays support negative indexing, where -1 refers to the last element, -2 refers to the second-last element, and so on.

By using these indexing techniques, you can efficiently access and manipulate elements or subsets of elements within NumPy arrays.

Array Transposition

Array transposition in NumPy refers to changing the shape and dimensions of an array by rearranging its axes. Transposition can be useful for various purposes, such as changing the row-major order to column-major order or for performing matrix operations. Here are some ways to transpose arrays in NumPy:

1. **Using the `T` attribute**: The `T` attribute can be used to transpose a 2D array by swapping its rows and columns.

```python

import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

transposed_arr = arr.T  # Transposes the array

# Output:
```

```
# array([[1, 4],

#        [2, 5],

#        [3, 6]])
```

2. **Using the `transpose` function**: The `transpose` function can be used to transpose an array by specifying the desired axes order.

```python
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

transposed_arr = np.transpose(arr)  # Transposes the array

# Output:

# array([[1, 4],

#        [2, 5],

#        [3, 6]])
```

3. **Using the `swapaxes` function**: The `swapaxes` function allows you to swap the positions of two axes in an array, effectively transposing it.

```python
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

transposed_arr = np.swapaxes(arr, 0, 1)  # Transposes the array by swapping axes 0 and 1

# Output:

# array([[1, 4],

#        [2, 5],

#        [3, 6]])
```

Note that transposing a 1D array has no effect, as it is already a single dimension.

Transposing an array does not create a new copy of the data; it simply changes the view of the array. Thus, transposing large arrays is efficient in terms of memory usage.

Array transposition can be particularly useful when performing operations such as matrix multiplication, calculating dot products, or reshaping arrays to match desired dimensions for computations.

Universal Array Function

Universal functions, also known as ufuncs, are functions in NumPy that operate element-wise on arrays, performing fast and efficient computations. Ufuncs provide a convenient and optimized way to apply mathematical operations, such as trigonometric functions, logarithmic functions, exponentiation, and more, to entire arrays without the need for explicit looping. Here are some examples of universal functions in NumPy:

1. **Mathematical Functions**: NumPy provides a wide range of mathematical functions as ufuncs, including:

- `np.sin()`, `np.cos()`, `np.tan()`: Trigonometric functions (sine, cosine, tangent)

- `np.exp()`: Exponential function

- `np.log()`, `np.log10()`, `np.log2()`: Logarithmic functions (natural logarithm, base 10 logarithm, base 2 logarithm)

- `np.sqrt()`: Square root function

- `np.power()`: Exponentiation

Example:

```python

import numpy as np

arr = np.array([1, 2, 3])

# Applying mathematical functions

result1 = np.sin(arr)  # Applies the sine function element-wise

result2 = np.exp(arr)  # Applies the exponential function element-wise

result3 = np.log10(arr)  # Applies the base 10 logarithm element-wise
```

```
print(result1)

print(result2)

print(result3)
```

2. **Arithmetic Operations**: Ufuncs can also be used for element-wise arithmetic operations, such as addition, subtraction, multiplication, division, and more.

Example:

```python
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

# Element-wise arithmetic operations

result1 = np.add(arr1, arr2)  # Element-wise addition

result2 = np.multiply(arr1, arr2)  # Element-wise multiplication

result3 = np.divide(arr2, arr1)  # Element-wise division

print(result1)

print(result2)

print(result3)
```

3. **Comparison Functions**: NumPy ufuncs also provide comparison functions to perform element-wise comparisons between arrays or between an array and a scalar value.

Example:

```python
import numpy as np
```

```
arr = np.array([1, 2, 3])

# Element-wise comparisons

result1 = np.greater(arr, 2)  # Element-wise greater than comparison

result2 = np.less_equal(arr, 2)  # Element-wise less than or equal to comparison

print(result1)

print(result2)
```

These are just a few examples of universal functions in NumPy. Ufuncs provide a powerful and efficient way to apply operations to arrays in a concise and vectorized manner, without the need for explicit looping. They are essential tools for performing numerical computations and data manipulation in NumPy.

Array Processing

Array processing in NumPy involves performing operations on arrays to manipulate, transform, and analyze data efficiently. NumPy provides a wide range of functions and methods for array processing. Here are some common array processing techniques:

1. **Element-wise Operations**: NumPy allows you to perform element-wise operations on arrays, where the operations are applied to each element individually. This includes arithmetic operations (addition, subtraction, multiplication, division), mathematical functions (sin, cos, exp, log), and comparison operations.

```python
import numpy as np

arr = np.array([1, 2, 3])

# Element-wise operations

result1 = arr + 2  # Adds 2 to each element of the array

result2 = np.sin(arr)  # Applies sine function to each element of the array

result3 = arr > 2  # Performs element-wise comparison

print(result1)
```

```
print(result2)

print(result3)
```

2. **Aggregation Functions**: NumPy provides functions for aggregating data from arrays, such as computing the sum, mean, maximum, minimum, standard deviation, and more. These functions can operate on the entire array or along a specific axis.

```python
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

# Aggregation functions

result1 = np.sum(arr)  # Computes the sum of all elements in the array

result2 = np.mean(arr, axis=0)  # Computes the mean along the rows (axis=0)

result3 = np.max(arr, axis=1)  # Computes the maximum along the columns (axis=1)

print(result1)

print(result2)

print(result3)
```

3. **Array Manipulation**: NumPy provides functions for reshaping, concatenating, and splitting arrays, allowing you to transform and combine arrays in various ways.

```python
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6]])

# Array manipulation

result1 = np.reshape(arr1, (4,))  # Reshapes the array to (4,) shape
```

```
result2 = np.concatenate((arr1, arr2), axis=0)  # Concatenates arrays vertically

result3 = np.split(arr1, 2, axis=1)  # Splits the array into two along the columns

print(result1)

print(result2)

print(result3)
```

4. **Array Broadcasting**: NumPy supports array broadcasting, which allows for performing operations on arrays with different shapes by automatically aligning and extending dimensions. This simplifies computations and avoids the need for explicit loops.

```python
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([[4], [5], [6]])


# Broadcasting

result = arr1 + arr2  # Adds arr1 to each row of arr2

print(result)
```

These are just a few examples of array processing techniques in NumPy. NumPy provides a comprehensive set of functions and methods for manipulating and analyzing arrays, making it a powerful tool for numerical computations and data analysis.

Array Input and Output

NumPy provides functions for inputting and outputting arrays to/from files. These functions allow you to save arrays to disk and load them back into memory. Here are the main functions for array input and output in NumPy:

1. **`np.save()` and `np.savez()`**: These functions are used to save a single array or multiple arrays into a binary `.npy` or `.npz` file, respectively.

```python

import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

# Saving arrays to file

np.save('array1.npy', arr1)  # Saves arr1 to a file named 'array1.npy'

np.savez('array2.npz', arr1=arr1, arr2=arr2)  # Saves arr1 and arr2 to a file named 'array2.npz' with named arrays

```

2. **`np.load()`**: This function is used to load arrays from a `.npy` or `.npz` file.

```python

import numpy as np

# Loading arrays from file

arr1 = np.load('array1.npy')  # Loads the array from 'array1.npy'

data = np.load('array2.npz')  # Loads the arrays from 'array2.npz'

arr2 = data['arr2']  # Accesses the 'arr2' array from the loaded data

```

3. **Text Files**: NumPy also provides functions to read from and write to text files. The `np.loadtxt()` function can be used to load data from a text file, and the `np.savetxt()` function can be used to save an array to a text file.

```python

import numpy as np

# Loading from a text file

data = np.loadtxt('data.txt')  # Loads data from 'data.txt'

# Saving to a text file

np.savetxt('results.txt', data)  # Saves the array 'data' to 'results.txt'

```

4. **Comma-Separated Values (CSV)**: NumPy provides functions to read and write CSV files using `np.genfromtxt()` and `np.savetxt()`. These functions allow you to handle CSV files with different delimiters, header options, and data types.

```python

import numpy as np

# Loading from a CSV file

data = np.genfromtxt('data.csv', delimiter=',', skip_header=1)  # Loads data from 'data.csv', skipping the header row

# Saving to a CSV file

np.savetxt('results.csv', data, delimiter=',', header='x,y,z')  # Saves the array 'data' to 'results.csv' with header

```

These are some of the functions available in NumPy for array input and output. These functions provide flexibility in storing and retrieving array data in various file formats, making it easy to persist and share array data between different applications and environments.

Pandas: Series, Data Frame, Panel

In Python, the Pandas library provides powerful data manipulation and analysis tools. It introduces three main data structures: Series, DataFrame, and Panel.

1. Series:

A Series is a one-dimensional labeled array that can hold any data type. It is similar to a column in a spreadsheet or a SQL table. Series objects consist of two main components: an index and data values. The index provides labels for each element in the Series, allowing for easy and efficient data retrieval. You can create a Series using the `pd.Series()` constructor.

Example:

```python

import pandas as pd

data = [10, 20, 30, 40, 50]

series = pd.Series(data)

print(series)
```

```

Output:

```

0 10

1 20

2 30

3 40

4 50

dtype: int64

```

2. DataFrame:

A DataFrame is a two-dimensional labeled data structure with columns of potentially different data types. It is similar to a table in a relational database or a spreadsheet. A DataFrame can be seen as a collection of Series objects, where each Series represents a column. You can create a DataFrame using the `pd.DataFrame()` constructor.

Example:

```python

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

    'Age': [25, 30, 35],

    'City': ['New York', 'London', 'Sydney']}

df = pd.DataFrame(data)



print(df)

```
```

Output:

```
    Name  Age     City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35    Sydney
```

3. Panel (Deprecated):

The Panel data structure in Pandas was designed to handle three-dimensional data. However, as of Pandas version 1.0.0, the Panel has been deprecated, and its usage is discouraged. Instead, it is recommended to use other data structures like the DataFrame, which can handle higher-dimensional data through the use of multi-indexing.

Note:

The Panel has been deprecated in favor of other data structures that offer more flexibility and better performance.

Index objects and Re-indexing

In Pandas, Index objects play a crucial role in aligning and labeling data in Series and DataFrame structures. An Index object represents the labels along a particular axis (either row or column) and provides methods for efficient data retrieval and alignment.

Here are some key aspects related to Index objects in Pandas:

1. Index Objects:

Index objects can be created independently or as part of Series and DataFrame structures. They are immutable and behave like an ordered set. The most common Index objects are `pd.Index`, `pd.RangeIndex`, and `pd.MultiIndex`. Index objects can contain labels of different types, such as integers, strings, or dates, depending on the data.

Example:

```python

import pandas as pd

index = pd.Index(['A', 'B', 'C', 'D'])

print(index)

```

Output:

```

Index(['A', 'B', 'C', 'D'], dtype='object')

```

2. Re-indexing:

Re-indexing is the process of creating a new Index object with a different order or set of labels. It allows you to align data to a new set of labels, add missing values, or reorder the existing data. The `reindex()` method is used to perform re-indexing in Pandas.

Example:

```python

import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data, index=['x', 'y', 'z'])

new_index = ['z', 'y', 'x']

df_reindexed = df.reindex(new_index)

print(df_reindexed)

```

Output:

```
```

```
  A  B
z 3  6
y 2  5
x 1  4
```

In the above example, the DataFrame `df` is re-indexed based on the new order specified by the `new_index` list.

Re-indexing can also be used to add missing labels by specifying the `fill_value` parameter or to interpolate missing values using methods like `ffill` or `bfill`.

These are some of the fundamental concepts related to Index objects and re-indexing in Pandas. Index objects and re-indexing are powerful tools for aligning, organizing, and manipulating data in Series and DataFrame structures.

Iteration

In Pandas, you can iterate over the elements in a Series or DataFrame using various methods. Additionally, you can sort the data in ascending or descending order based on one or more columns. Let's explore both concepts:

1. Iteration:

a. Iterating over Series:

To iterate over the elements of a Series, you can use the `iteritems()` method, which returns an iterator yielding index-label pairs.

Example:

```python
import pandas as pd

series = pd.Series([10, 20, 30, 40, 50])

for index, value in series.iteritems():
    print(f'Index: {index}, Value: {value}')
```

```

Output:

```

Index: 0, Value: 10

Index: 1, Value: 20

Index: 2, Value: 30

Index: 3, Value: 40

Index: 4, Value: 50

```

b. Iterating over DataFrame:

To iterate over the elements of a DataFrame, you can use the `iterrows()` method, which returns an iterator yielding index-row pairs as tuples.

Example:

```python

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

        'Age': [25, 30, 35],

        'City': ['New York', 'London', 'Sydney']}

df = pd.DataFrame(data)

for index, row in df.iterrows():

    print(f'Index: {index}')

    print(f'Row:\n{row}')

    print('---')

```
```

Output:

```

Index: 0

Row:

Name        Alice

Age          25

City      New York

Name: 0, dtype: object

---

Index: 1

Row:

Name       Bob

Age         30

City     London

Name: 1, dtype: object

---

Index: 2

Row:

Name      Charlie

Age          35

City       Sydney

Name: 2, dtype: object

---
```

Sorting

In Pandas, you can sort data in a DataFrame using the `sort_values()` method. This method allows you to sort the DataFrame based on one or more columns. Let's explore the sorting process in more detail:

1. Sorting a DataFrame by Single Column:

To sort a DataFrame by a single column, you can use the `sort_values()` method and specify the column name using the `by` parameter. By default, the sorting is done in ascending order, but you can change this behavior by setting the `ascending` parameter to `False` for descending order.

Example:

```python
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'London', 'Sydney']}

df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Age', ascending=True)

print(sorted_df)
```

Output:

```
      Name  Age     City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35    Sydney
```

In the above example, the DataFrame is sorted in ascending order based on the 'Age' column.

2. Sorting a DataFrame by Multiple Columns:

You can also sort a DataFrame based on multiple columns. The `sort_values()` method allows you to pass a list of column names to the `by` parameter. The DataFrame will be sorted first by the first column, then by the second column, and so on.

Example:

```python

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

        'Age': [30, 25, 30],

        'City': ['New York', 'London', 'Sydney']}

df = pd.DataFrame(data)

sorted_df = df.sort_values(by=['Age', 'Name'], ascending=[True, False])

print(sorted_df)

```

Output:

```

      Name  Age      City

1      Bob   25    London

0    Alice   30  New York

2  Charlie   30    Sydney

```

In the above example, the DataFrame is sorted first by the 'Age' column in ascending order, and then by the 'Name' column in descending order.

3. Sorting by Index:

If you want to sort the DataFrame based on the index, you can use the `sort_index()` method.

Example:

```python
```

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'London', 'Sydney']}

df = pd.DataFrame(data)

sorted_df = df.sort_index(ascending=False)

print(sorted_df)
```

Output:

```
      Name  Age     City
2  Charlie   35    Sydney
1      Bob   30    London
0    Alice   25  New York
```

In the above example, the DataFrame is sorted based on the index in descending order using the `sort_index()` method.

These are the primary methods for sorting data in a Pandas DataFrame. Sorting can help you organize and analyze data in a desired order, making it easier to derive insights and perform further analysis.

Matplotlib

Matplotlib is a popular Python library used for data visualization. It provides a wide range of functions and tools for creating high-quality plots, charts, and figures. Matplotlib is highly customizable and allows you to create visualizations for various purposes, such as exploring data, presenting findings, or creating publication-ready graphics.

Here's a brief overview of how Matplotlib works and some of its key features:

Installation:

Matplotlib can be installed using pip, the package installer for Python. Open a terminal or command prompt and run the following command:

```

pip install matplotlib

```

Basic Usage:

To start using Matplotlib, you need to import it in your Python script or Jupyter Notebook:

```python

import matplotlib.pyplot as plt

```

`pyplot` is a subpackage within Matplotlib that provides a MATLAB-like interface for creating plots.

Creating a Simple Plot:

The most common type of plot in Matplotlib is a line plot. You can create a basic line plot by providing the x and y coordinates of the data points and then using the `plot()` function:

```python

import matplotlib.pyplot as plt

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

y = [2, 4, 6, 8, 10]

plt.plot(x, y)
```

plt.show()

```

This code will generate a simple line plot with the x-axis values `[1, 2, 3, 4, 5]` and the corresponding y-axis values `[2, 4, 6, 8, 10]`.

Customizing Plots:

Matplotlib provides a wide range of customization options to enhance your plots. You can add labels to axes, a title, legends, change line colors and styles, adjust axis limits, add gridlines, and much more. Here's an example that demonstrates some of these customizations:

```python

import matplotlib.pyplot as plt

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

y = [2, 4, 6, 8, 10]

plt.plot(x, y, color='red', linestyle='--', marker='o', label='data')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Simple Plot')

plt.legend()

plt.grid(True)

plt.show()

```

In this code, we changed the line color to red, the line style to dashed (`--`), and added circular markers at the data points. We also added labels to the x and y axes, a title, a legend, and gridlines.

Matplotlib supports various types of plots, including bar plots, scatter plots, histograms, pie charts, and more. You can explore the official Matplotlib documentation and gallery for more examples and detailed information on different plot types and customization options.

Remember to import the necessary libraries at the beginning of your script or notebook, such as `matplotlib.pyplot` for plotting and `numpy` for numerical calculations if needed.
```

Python for Data Visualization

Python is a powerful programming language that offers numerous libraries and tools for data visualization. These libraries provide a wide range of functions and capabilities for creating visually appealing and informative plots, charts, and graphs. Here are some popular Python libraries for data visualization:

1. Matplotlib: Matplotlib, as mentioned earlier, is a widely used library for creating static, animated, and interactive visualizations. It offers a MATLAB-like interface and provides extensive customization options.

2. Seaborn: Seaborn is built on top of Matplotlib and provides a higher-level interface for creating statistical graphics. It focuses on enhancing the visual appeal of plots and provides functions for creating attractive and informative statistical visualizations.

3. Plotly: Plotly is a library that enables the creation of interactive and dynamic visualizations. It offers a range of chart types, including scatter plots, line plots, bar charts, heatmaps, and 3D plots. Plotly visualizations can be embedded in web applications or shared online.

4. Pandas: Pandas is primarily a data manipulation library, but it also provides basic plotting capabilities. It allows you to create plots directly from Pandas data structures, such as DataFrames and Series. Pandas uses Matplotlib under the hood, so you can further customize the plots using Matplotlib functions.

5. Bokeh: Bokeh is a library for creating interactive visualizations for modern web browsers. It emphasizes interactivity and can handle large and streaming datasets. Bokeh supports various plot types and provides interactive tools, such as zooming, panning, and hovering.

6. Altair: Altair is a declarative statistical visualization library that focuses on simplicity and clarity. It allows you to create visualizations by specifying the intent of the visualization rather than the low-level details. Altair is built on top of the Vega and Vega-Lite libraries and supports interactive exploration.

These libraries offer different approaches to data visualization, and the choice depends on your specific requirements and preferences. It's worth exploring each library's documentation and examples to understand their capabilities and find the one that best suits your needs. Additionally, Jupyter Notebooks are a popular tool for data visualization in Python, as they provide an interactive environment for creating, exploring, and sharing visualizations alongside code and text explanations.

Visualization Section

Certainly! Here's an example of how you can create a visualization using Matplotlib in Python:

```python

import matplotlib.pyplot as plt
```

```
# Sample data

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

y = [2, 4, 6, 8, 10]


# Create a line plot

plt.plot(x, y)


# Add labels and 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 simple line plot using the `plot()` function from Matplotlib. We provide the x and y values as lists, and Matplotlib automatically connects the data points with lines.

We then add labels to the x and y axes using `xlabel()` and `ylabel()`, respectively. The `title()` function is used to set the title of the plot. Finally, we display the plot using `show()`.

This code will generate a window or output the plot in a Jupyter Notebook, showing a line plot with the x-axis values `[1, 2, 3, 4, 5]` and the corresponding y-axis values `[2, 4, 6, 8, 10]`. You can further customize the plot by exploring additional options and functions provided by Matplotlib.

Remember to import `matplotlib.pyplot` at the beginning of your script or notebook to use Matplotlib functions conveniently.

Feel free to experiment with different types of plots, customize the appearance, and explore other libraries like Seaborn, Plotly, or Pandas for additional data visualization options in Python.

Sklearn: loading of dataset

In scikit-learn (sklearn), there are several built-in datasets that you can load and use for machine learning tasks. These datasets are accessible through the `sklearn.datasets` module. Here's an example of how you can load a dataset using scikit-learn:

```python

from sklearn import datasets

# Load the Iris dataset

iris = datasets.load_iris()

# Access the features and target variables

X = iris.data  # Features

y = iris.target  # Target variable

# Print the shape of the data

print("Shape of features:", X.shape)

print("Shape of target variable:", y.shape)

```

In this example, we load the Iris dataset using the `load_iris()` function from the `datasets` module. The Iris dataset is a commonly used dataset in machine learning, and it contains measurements of different characteristics of iris flowers along with their corresponding species.

After loading the dataset, we can access the features (input variables) and the target variable (output variable). In this case, `X` represents the features (a NumPy array), and `y` represents the target variable (a 1-dimensional NumPy array).

We then print the shape of the data to verify that it has been loaded correctly. The `shape` attribute of a NumPy array returns the dimensions of the array (number of samples, number of features).

You can similarly load other built-in datasets from scikit-learn by replacing `load_iris()` with the appropriate dataset loading function, such as `load_digits()`, `load_boston()`, `load_wine()`, etc. These datasets cover various domains and can be used for classification, regression, clustering, and other machine learning tasks.

Additionally, scikit-learn also provides functions to split the data into training and testing sets, preprocess the data, and perform various other operations necessary for machine learning.

learning and predicting

Once you have loaded a dataset in scikit-learn, you can proceed with training a machine learning model on the data and using it to make predictions. Here's an example that demonstrates the basic steps of learning from the data and making predictions using scikit-learn:

```python

from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

# Load the Iris dataset

iris = datasets.load_iris()

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)

# Create an instance of the K-nearest neighbors classifier

knn = KNeighborsClassifier()

# Train the model on the training data

knn.fit(X_train, y_train)

# Make predictions on the test data

y_pred = knn.predict(X_test)

# Print the predicted labels and the true labels

print("Predicted labels:", y_pred)

print("True labels:", y_test)

```

In this example, we follow these steps:

1. Load the Iris dataset using `datasets.load_iris()`.

2. Split the data into training and testing sets using `train_test_split()`. The `test_size` parameter determines the proportion of the data to be used for testing, and `random_state` ensures reproducibility of the split.

3. Create an instance of a machine learning model, in this case, `KNeighborsClassifier()`, which is a classifier based on the k-nearest neighbors algorithm.

4. Train the model on the training data using the `fit()` method. This step involves learning patterns and relationships in the training data.

5. Make predictions on the test data using the `predict()` method. The model uses the learned patterns to predict the target variable for the test instances.

6. Finally, we print the predicted labels and the true labels to evaluate the performance of the model.

Note that this is a simple example, and in practice, you would typically perform more steps, such as data preprocessing, feature engineering, model evaluation, and hyperparameter tuning. Scikit-learn provides a wide range of tools and functionalities to support these tasks.

You can explore the scikit-learn documentation for more information on different algorithms, model evaluation techniques, and additional functionalities available for machine learning tasks.

Model Persistence

Model persistence refers to the process of saving a trained machine learning model to disk and loading it later for reuse. This allows you to save time and computational resources by avoiding the need to retrain the model from scratch each time you want to make predictions. Scikit-learn provides tools for model persistence through its `joblib` module. Here's an example of how you can save and load a scikit-learn model:

```python

from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

import joblib

# Load the Iris dataset

iris = datasets.load_iris()

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)

# Create an instance of the logistic regression model

model = LogisticRegression()
```

```
# Train the model on the training data

model.fit(X_train, y_train)

# Save the model to disk

joblib.dump(model, 'model.pkl')

# Load the model from disk

loaded_model = joblib.load('model.pkl')

# Make predictions using the loaded model

predictions = loaded_model.predict(X_test)

# Print the predictions

print(predictions)
```

In this example, we follow these steps:

1. Load the Iris dataset and split it into training and testing sets, just like in the previous example.

2. Create an instance of a machine learning model. In this case, we use `LogisticRegression()` as an example.

3. Train the model on the training data.

4. Save the trained model to disk using `joblib.dump()`. The first argument is the model object to be saved, and the second argument is the file path where the model will be saved.

5. Load the saved model from disk using `joblib.load()`. The argument is the file path of the saved model.

6. Use the loaded model to make predictions on new data. In this case, we use the test set `X_test`.

7. Finally, we print the predictions made by the loaded model.

By saving the model using `joblib.dump()` and loading it later with `joblib.load()`, you can reuse the trained model without having to retrain it every time you need to make predictions.

Remember to install `joblib` if you haven't done so already, using the command `pip install joblib`. Additionally, you can choose other file formats like pickle (`pickle.dump()` and `pickle.load()`) for model persistence, but `joblib` is often preferred for scikit-learn models due to its efficient handling of large NumPy arrays.