

SHRIMATI INDIRA GANDHI COLLEGE

**Affiliated to Bharathidasan University| Nationally Accredited at 'A' Grade(3rd Cycle) by
NAAC**

An ISO 9001:2015 Certified Institution

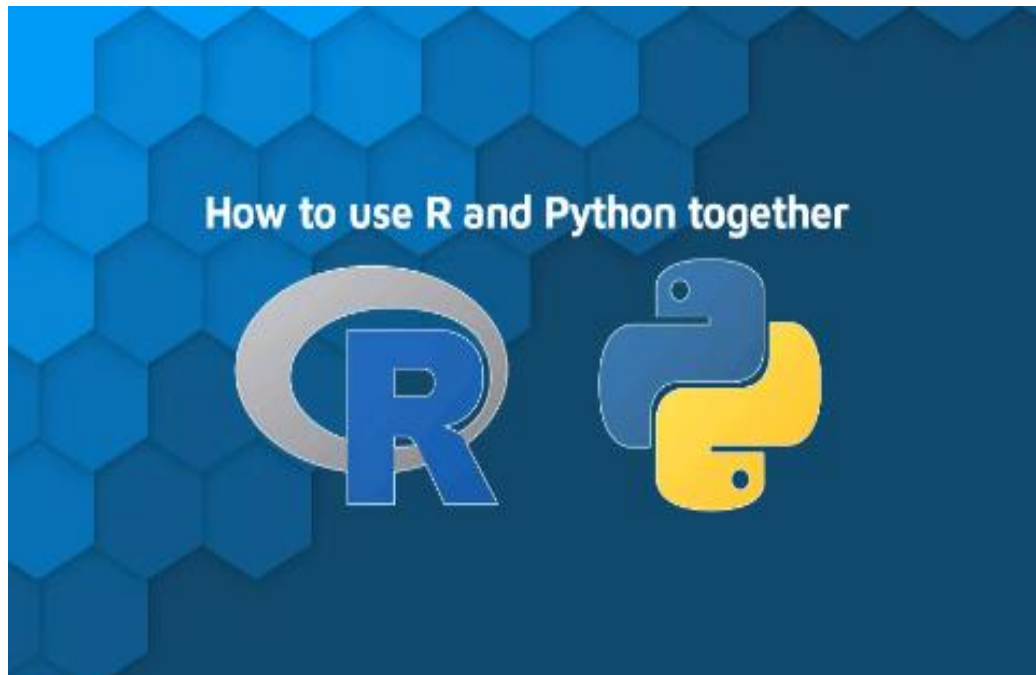
Thiruchirappalli

STUDY MATERIAL

Problem Solving using Python and R (P22CSCC12)



**DEPARTMENT OF COMPUTER SCIENCE, INFORMATION
TECHNOLOGY AND COMPUTER APPLICATIONS**



Prepared by,

MS.S.S.NACHIYA , M.C.A., M.Phil.,UGC-NET

ASST. PROF. IN COMPUTER SCIENCE,

SHRIMATI INDIRA GANDHI COLLEGE,

TIRUCHIRAPPALLI - 2

First Year

**CORE COURSE II
PROBLEM SOLVING USING PYTHON
AND R**

Semester I

Code:

(Theory)

Credit: 5

COURSE OBJECTIVES:

- To understand Computational thinking using Python.
- To develop simple Python programs for solving problems.
- To make students exercise the fundamentals of statistical analysis in R environment.

UNIT – 1 INTRODUCTION TO PYTHON:

Introduction – Python overview – Getting started – Comments – Python identifiers – Reserved keywords – Variables – Standard data types – Operators – Statements and Expressions – String operations – Boolean expressions. Control Statements: The for loop – while statement – if-elif-else statement – Input from keyboard. Functions: Introduction – Built-in functions – User defined functions – Function Definition – Function Call - Type conversion – Type coercion – Python recursivefunction.

UNIT – II STRINGS:

Strings –Compound data type – len function – String slices – String traversal – Escape characters – String formatting operator – String formatting functions. Tuples: Tuples – Creating tuples – Accessing values in tuples – Tuple assignment – Tuples as return values – Basic tuple operations – Built-in tuple functions. Lists: Values and accessing elements – Traversing a list – Deleting elements from list – Built-in list operators & methods. Dictionaries: Creating dictionary – Accessing values in dictionary – Updating dictionary – Deleting elements from dictionary – Operations in dictionary - Built-in dictionary methods.

UNIT – III FILES AND EXCEPTIONS:

Introduction to File Input and Output - Writing Structures to a File - Using loops to process files Processing Records - Exception. Classes and Objects in Python: Overview of OOP – Data encapsulation – Polymorphism – Class definition – Creating objects – Inheritance – Multiple inheritances – Method overriding – Data encapsulation – Data hiding.

UNIT – IV DATA MANIPULATION TOOLS & SOFTWARES:

Numpy: Installation - Ndarray - Basic Operations -Indexing, Slicing, and Iterating - Shape Manipulation - Array Manipulation - Structured Arrays -Reading and Writing Array Data on Files. Pandas: The pandas Library: An Introduction - Installation - Introduction to pandas Data Structures - Operations between Data Structures - Function Application and Mapping - Sorting and Ranking - Correlation and Covariance - —Not a Number Data - Hierarchical Indexing and Leveling – Reading and Writing Data: CSV or Text File - HTML Files – Microsoft Excel Files.

UNIT – V PROGRAMMING WITH R:

Variables - Vector, matrix, arrays – List – Data Frames – Functions – Strings – Factors – Loops – Packages –Date and Time – Files - Making packages

Unit -I

Introduction to Python

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Basic Syntax

There is no use of curly braces or semicolon in Python programming language. It is English-like language. But Python uses the indentation to define a block of code. Indentation is nothing but adding whitespace before the statement when it is needed. **For example -**

```
def func():  
    statement 1  
    statement 2  
    .....  
    .....  
    statement N
```

note

Python is a case-sensitive language, which means that uppercase and lowercase letters are treated differently. For example, 'name' and 'Name' are two different variables in Python.

Advantages of Python

Python provides many useful features to the programmer. These features make it the most popular and widely used language. We have listed below few-essential features of Python.

- **Easy to use and Learn:** Python has a simple and easy-to-understand syntax, unlike traditional languages like C, C++, Java, etc., making it easy for beginners to learn.
- **Expressive Language:** It allows programmers to express complex concepts in just a few lines of code or reduces Developer's Time.
- **Interpreted Language:** Python does not require compilation, allowing rapid development and testing. It uses Interpreter instead of Compiler.
- **Object-Oriented Language:** It supports object-oriented programming, making writing reusable and modular code easy.
- **Open Source Language:** Python is open source and free to use, distribute and modify.
- **Extensible:** Python can be extended with modules written in C, C++, or other languages.
- **Learn Standard Library:** Python's standard library contains many modules and functions that can be used for various tasks, such as string manipulation, web programming, and more.

- **GUI Programming Support:** Python provides several GUI frameworks, such as Tkinter and PyQt, allowing developers to create desktop applications easily.
- **Integrated:** Python can easily integrate with other languages and technologies, such as C/C++, Java, and .NET.
- **Embeddable:** Python code can be embedded into other applications as a scripting language.
- **Dynamic Memory Allocation:** Python automatically manages memory allocation, making it easier for developers to write complex programs without worrying about memory management.
- **Wide Range of Libraries and Frameworks:** Python has a vast collection of libraries and frameworks, such as NumPy, Pandas, Django, and Flask, that can be used to solve a wide range of problems.
- **Versatility:** Python is a universal language in various domains such as web development, machine learning, data analysis, scientific computing, and more.
- **Large Community:** Python has a vast and active community of developers contributing to its development and offering support. This makes it easy for beginners to get help and learn from experienced developers.
- **Career Opportunities:** Python is a highly popular language in the job market. Learning Python can open up several career opportunities in data science, artificial intelligence, web development, and more.
- **High Demand:** With the growing demand for automation and digital transformation, the need for Python developers is rising. Many industries seek skilled Python developers to help build their digital infrastructure.
- **Increased Productivity:** Python has a simple syntax and powerful libraries that can help developers write code faster and more efficiently. This can increase productivity and save time for developers and organizations.
- **Big Data and Machine Learning:** Python has become the go-to language for big data and machine learning. Python has become popular among data scientists and machine learning engineers with libraries like NumPy, Pandas, Scikit-learn, TensorFlow, and more.

Where is Python used?

Python is a general-purpose, popular programming language, and it is used in almost every technical field. The various areas of Python use are given below.

- **Data Science:** Data Science is a vast field, and Python is an important language for this field because of its simplicity, ease of use, and availability of powerful data analysis and visualization libraries like NumPy, Pandas, and Matplotlib.

- **Desktop Applications:** PyQt and Tkinter are useful libraries that can be used in GUI - Graphical User Interface-based Desktop Applications. There are better languages for this field, but it can be used with other languages for making Applications.
- **Console-based Applications:** Python is also commonly used to create command-line or console-based applications because of its ease of use and support for advanced features such as input/output redirection and piping.
- **Mobile Applications:** While Python is not commonly used for creating mobile applications, it can still be combined with frameworks like Kivy or BeeWare to create cross-platform mobile applications.
- **Software Development:** Python is considered one of the best software-making languages. Python is easily compatible with both from Small Scale to Large Scale software.
- **Artificial Intelligence:** AI is an emerging Technology, and Python is a perfect language for artificial intelligence and machine learning because of the availability of powerful libraries such as TensorFlow, Keras, and PyTorch.
- **Web Applications:** Python is commonly used in web development on the backend with frameworks like Django and Flask and on the front end with tools like JavaScript and HTML.
- **Enterprise Applications:** Python can be used to develop large-scale enterprise applications with features such as distributed computing, networking, and parallel processing.
- **3D CAD Applications:** Python can be used for 3D computer-aided design (CAD) applications through libraries such as Blender.
- **Machine Learning:** Python is widely used for machine learning due to its simplicity, ease of use, and availability of powerful machine learning libraries.
- **Computer Vision or Image Processing Applications:** Python can be used for computer vision and image processing applications through powerful libraries such as OpenCV and Scikit-image.
- **Speech Recognition:** Python can be used for speech recognition applications through libraries such as SpeechRecognition and PyAudio.
- **Scientific computing:** Libraries like NumPy, SciPy, and Pandas provide advanced numerical computing capabilities for tasks like data analysis, machine learning, and more.
- **Education:** Python's easy-to-learn syntax and availability of many resources make it an ideal language for teaching programming to beginners.
- **Testing:** Python is used for writing automated tests, providing frameworks like unit tests and pytest that help write test cases and generate reports.
- **Gaming:** Python has libraries like Pygame, which provide a platform for developing games using Python.

- **IoT:** Python is used in IoT for developing scripts and applications for devices like Raspberry Pi, Arduino, and others.
- **Networking:** Python is used in networking for developing scripts and applications for network automation, monitoring, and management.
- **DevOps:** Python is widely used in DevOps for automation and scripting of infrastructure management, configuration management, and deployment processes.
- **Finance:** Python has libraries like Pandas, Scikit-learn, and Statsmodels for financial modeling and analysis.
- **Audio and Music:** Python has libraries like Pyaudio, which is used for audio processing, synthesis, and analysis, and Music21, which is used for music analysis and generation.
- **Writing scripts:** Python is used for writing utility scripts to automate tasks like file operations, web scraping, and data processing.

How to Get Started With Python?

Python is a cross-platform programming language, which means that it can run on multiple platforms like Windows, macOS, Linux, and has even been ported to the Java and .NET virtual machines. It is free and open-source.

Even though most of today's Linux and Mac have Python pre-installed in it, the version might be out-of-date. So, it is always a good idea to install the most current version.

1. Run Python in Immediate mode

Once Python is installed, typing `python` in the command line will invoke the interpreter in immediate mode. We can directly type in Python code, and press Enter to get the output.

Try typing in `1 + 1` and press enter. We get `2` as the output. This prompt can be used as a calculator. To exit this mode, type `quit()` and press enter.

```
Microsoft Windows [Version 10.0.17134.648]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\ASUS>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more i
nformation.
>>> 1 + 1
2
>>> quit()

C:\Users\ASUS>_
```

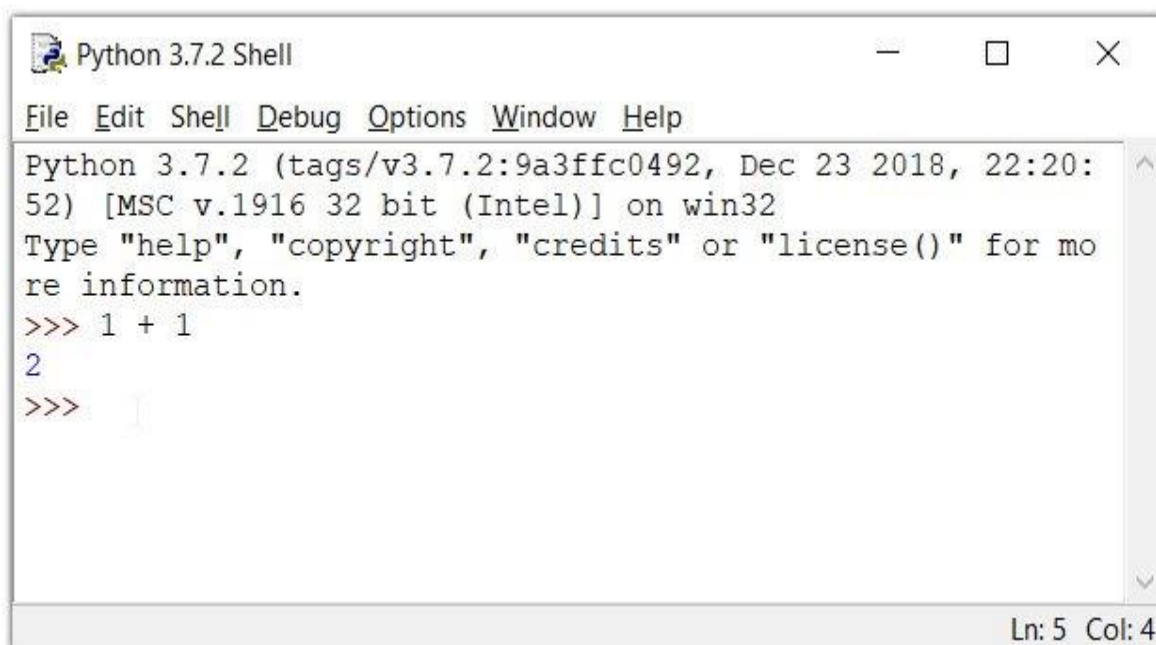
2. Run Python in the Integrated Development Environment (IDE)

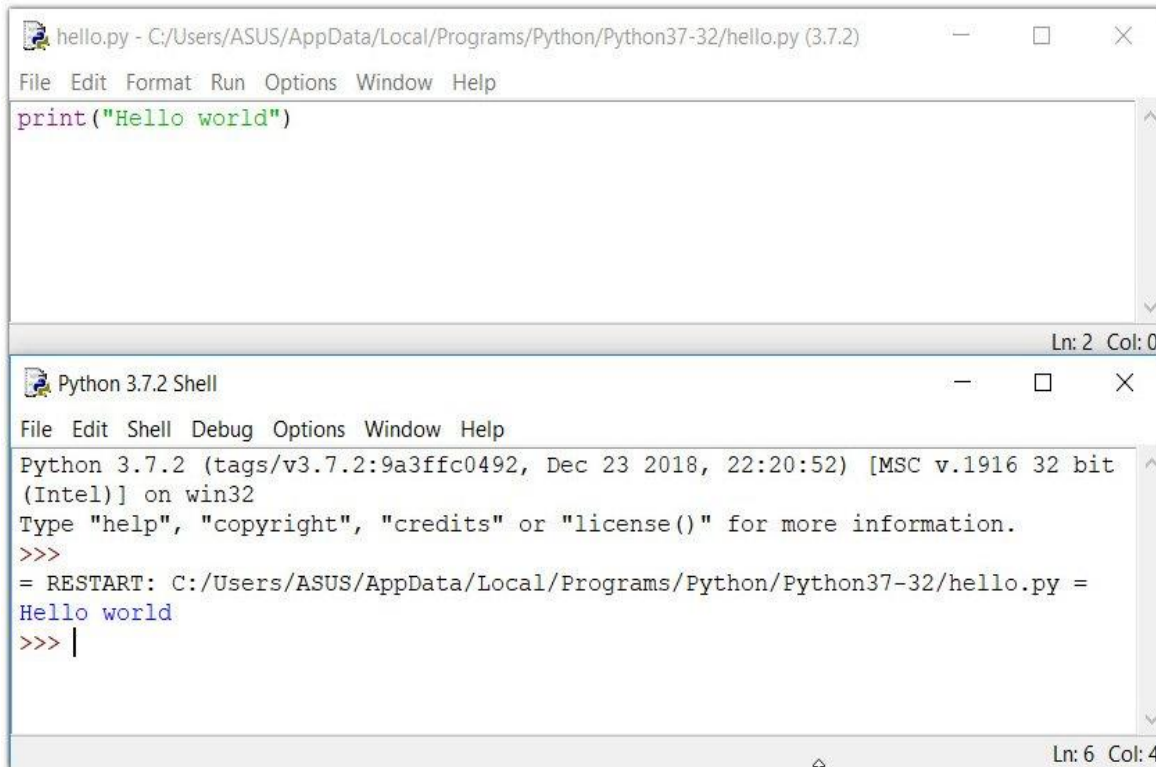
We can use any text editing software to write a Python script file.

We just need to save it with the `.py` extension. But using an IDE can make our life a lot easier. IDE is a piece of software that provides useful features like code hinting, syntax highlighting and checking, file explorers, etc. to the programmer for application development.

By the way, when you install Python, an IDE named **IDLE** is also installed. You can use it to run Python on your computer. It's a decent IDE for beginners.

When you open IDLE, an interactive Python Shell is opened.





```
hello.py - C:/Users/ASUS/AppData/Local/Programs/Python/Python37-32/hello.py (3.7.2)
File Edit Format Run Options Window Help
print("Hello world")
Ln: 2 Col: 0

Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/ASUS/AppData/Local/Programs/Python/Python37-32/hello.py =
Hello world
>>> |
Ln: 6 Col: 4
```

Now you can create a new file and save it with **.py** extension. For example, **hello.py**
Write Python code in the file and save it. To run the file, go to **Run > Run Module** or simply click **F5**.

Python Comments

Comments in Python are the lines in the code that are ignored by the interpreter during the execution of the program. Comments enhance the readability of the code and help the programmers to understand the code very carefully.

There are three types of comments in [Python](#):

- Single line Comments
- Multiline Comments
- Docstring Comments

Single Line Comments

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

Example

Following is an example of a single line comment in Python:

```
# This is a single line comment in python
```

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

This produces the following result –

Hello, World!

You can type a comment on the same line after a statement or expression –

```
name = "Madisetti" # This is again comment
```

Multi-Line Comments

Python does not provide a direct way to comment multiple line. You can comment multiple lines as follows –

```
# This is a comment.
```

```
# This is a comment, too.
```

```
# This is a comment, too.
```

```
# I said that already.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as a **multiline comments:**

```
'''
```

```
This is a multiline
```

Example

Following is the example to show the usage of multi-line comments:

```
'''
```

```
This is a multiline  
comment.
```

```
'''
```

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

This produces the following result –

Hello, World!

Docstring Comments

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.

```
def add(a, b):  
    """Function to add the value of a and b"""  
    return a+b  
print(add.__doc__)
```

This produces the following result –

Function to add the value of a and b

Identifiers in Python

Identifier is a user-defined name given to a variable, function, class, module, etc. The identifier is a combination of character digits and an underscore. They are case-sensitive i.e., 'num' and 'Num' and 'NUM' are three different identifiers in python.

Rules for Naming Python Identifiers

- It cannot be a reserved python keyword.
- It should not contain white space.
- It can be a combination of A-Z, a-z, 0-9, or underscore.
- It should start with an alphabet character or an underscore (`_`).
- It should not contain any special character other than an underscore (`_`).

Some Valid and Invalid Identifiers in Python

Valid Identifiers

Invalid Identifiers

score	@core
return_value	return
highest_score	highest score
name1	1name
convert_to_string	convert to_string

Python Keywords and Identifiers Examples

Example 1: Example of and, or, not, True, False keywords.

Python

```
print("example of True, False, and, or, not keywords")

# compare two operands using and operator
print(True and True)

# compare two operands using or operator
print(True or False)

# use of not operator
print(not False)
```

Output:

example of True, False, and, or, not keywords

True

True

True

Other examples

Language='Python'

Continue='Python'

Reserved words

Python Keywords are some predefined and reserved words in python that have special meanings. Keywords are used to define the syntax of the coding. The keyword cannot be used as an identifier, function, or variable name. All the keywords in python are written in lowercase except True and False. There are 35 keywords in Python 3.11.

In python, there is an inbuilt [keyword module](#) that provides an [iskeyword\(\) function](#) that can be used to check whether a given string is a valid keyword or not. Furthermore we can check the name of the keywords in Python by using the kwlist attribute of the keyword module.

Rules for Keywords in Python

- Python keywords cannot be used as identifiers.
- All the keywords in python should be in lowercase except True and False.

List of Python Keywords

Keywords	Description
and	This is a logical operator which returns true if both the operands are true else returns false.
or	This is also a logical operator which returns true if anyone operand is true else returns false.
not	This is again a logical operator it returns True if the operand is false else returns false.
if	This is used to make a conditional statement.
elif	Elif is a condition statement used with an if statement. The elif statement is executed if the previous conditions were not true.
else	Else is used with if and elif conditional statements. The else block is executed if the given condition is not true.
for	This is used to create a loop.
while	This keyword is used to create a while loop.

Keywords	Description
break	This is used to terminate the loop.
as	This is used to create an alternative.
def	It helps us to define functions.
lambda	It is used to define the anonymous function.
pass	This is a null statement which means it will do nothing.
return	It will return a value and exit the function.
True	This is a boolean value.
False	This is also a boolean value.
try	It makes a try-except statement.
with	The with keyword is used to simplify exception handling.
assert	This function is used for debugging purposes. Usually used to check the correctness of code
class	It helps us to define a class.
continue	It continues to the next iteration of a loop
del	It deletes a reference to an object.
except	Used with exceptions, what to do when an exception occurs
finally	Finally is used with exceptions, a block of code that will be executed no matter if there is an exception or not.
from	It is used to import specific parts of any module.

Keywords	Description
global	This declares a global variable.
import	This is used to import a module.
in	It's used to check whether a value is present in a list, range, tuple, etc.
is	This is used to check if the two variables are equal or not.
nonlocal	It's declared a non-local variable.
raise	This raises an exception.
yield	It ends a function and returns a generator.
async	It is used to create asynchronous coroutines.
await	It releases the flow of control back to the event loop.
none	This is a special constant used to denote a null value or void. It's important to remember, 0, any empty container(e.g empty list) do not compute to None.

Variables in Python

Python Variable is containers that store values. Python is not “statically typed”. We do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it. A Python variable is a name given to a memory location. It is the basic unit of storage in a program.

Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

Output:

>3

>Jhon

Note:

- The value stored in a variable can be changed during program execution.
- A Variables in Python is only a name given to a memory location, all the operations done on the variable effects that memory location.

Rules for Python variables

- A Python variable name must start with a letter or the underscore character.
- A Python variable name cannot start with a number.
- A Python variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable in Python names are case-sensitive (name, Name, and NAME are three different variables).
- The reserved words(keywords) in Python cannot be used to name the variable in Python.

Example

```
# valid variable name
geeks =1
Geeks =2
Ge_e_ks=5
_geeks =6
geeks_ =7
_GEEKS_ =8
print(geeks, Geeks, Ge_e_ks)
print(_geeks, geeks_, _GEEKS_)
```

Output:

1 2 5

6 7 8

Variables Assignment in Python

Here, we have assigned a number, a floating point number, and a string to a variable such as age, salary, and name.

Example

```
# An integer assignment
```

```
age =45
```

```
# A floating point
```

```
salary =1456.8
```

```
# A string
```

```
name ="John"
```

```
print(age)
```

```
print(salary)
```

```
print(name)
```

Output:

```
45
```

```
1456.8
```

```
John
```

Redeclaring variables in Python

It can re-declare the Python variable once we have declared the variable already.

Example

```
# declaring the var
```

```
Number =100
```

```
# display
```

```
print("Before declare: ", Number)
```

```
# re-declare the var
```

```
Number =120.3
```

```
print("After re-declare:", Number)
```

Output:

Before declare: 100

After re-declare: 120.3

Python Assign Values to Multiple Variables

Python allows assigning a single value to several variables simultaneously with “=” operators.

For example:

```
a =b =c =10
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

Output:

10

10

10

Assigning different values to multiple variables

Python allows adding different values in a single line with “,” operators.

```
a, b, c =1, 20.2, "GeeksforGeeks"
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

Output:

1

20.2

GeeksforGeeks

Python Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example –

```
var1 = 1
```

```
var2 = 10
```

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

Python Lists

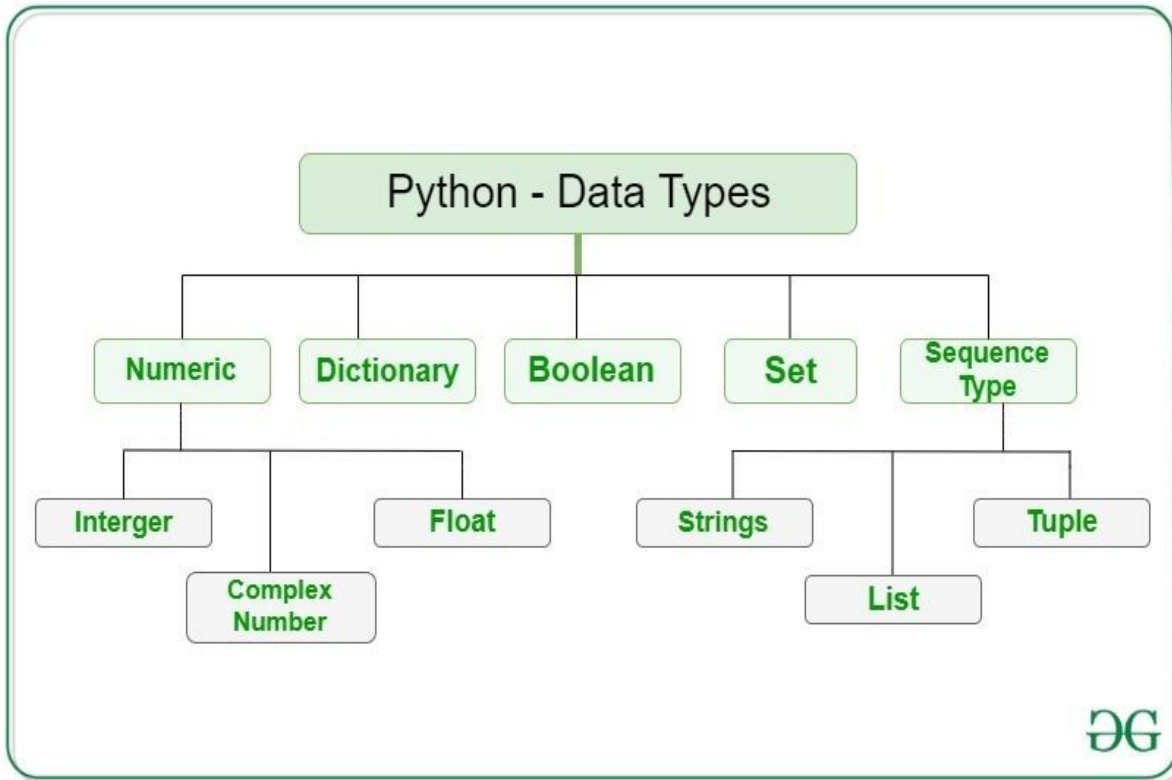
Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.



Numbers

Numeric values are stored in numbers. The whole number, float, and complex qualities have a place with a Python Numbers datatype. Python offers the `type()` function to determine a variable's data type. The `instance ()` capability is utilized to check whether an item has a place with a specific class.

- **Int:** Whole number worth can be any length, like numbers 10, 2, 29, - 20, - 150, and so on. An integer can be any length you want in Python. Its worth has a place with `int`.
- **Float:** Float stores drifting point numbers like 1.9, 9.902, 15.2, etc. It can be accurate to within 15 decimal places.
- **Complex:** An intricate number contains an arranged pair, i.e., $x + iy$, where x and y signify the genuine and non-existent parts separately. The complex numbers like $2.14j$, $2.0 + 2.3j$, etc.

Example

1. `a = 5`
2. `print("The type of a", type(a))`
3. `b = 40.5`
4. `print("The type of b", type(b))`
5. `c = 1+3j`
6. `print("The type of c", type(c))`
7. `print(" c is a complex number", isinstance(1+3j,complex))`

Output

The type of a <class 'int'>

The type of b <class 'float'>

The type of c <class 'complex'>

c is complex number: True

Set

Unordered objects are grouped together as a set. There cannot be any duplicates of any set element, and it must be immutable (cannot be changed).

Creation of set

The built-in `set()` method can be used to build sets with an iterable object or a series by wrapping the sequence behind curly brackets and separating them with a comma,. The elements in a set don't have to be of the same type; they might contain a variety of mixed data type values

Example

```
# Create a set from a list using the set() function
```

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

```
print(s)# Output: {1, 2, 3, 4, 5}
```

```
# Create a set using curly braces
```

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

```
print(s)# Output: {1, 2, 3, 4, 5}
```

Output

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

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

Boolean

True and False are the two default values for the Boolean type. These qualities are utilized to decide the given assertion valid or misleading. The class bool indicates this. False can be represented by the 0 or the letter "F," while true can be represented by any value that is not zero.

Example

1. `# Python program to check the boolean type`
2. `print(type(True))`
3. `print(type(False))`
4. `print(false)`

Output:

```
<class 'bool'>
```

```
<class 'bool'>
```

```
NameError: name 'false' is not defined
```

Sequence

The sequence in Python is an ordered grouping of related or dissimilar data types. Sequences enable the ordered and effective storage of several values. In Python, there are various sequence types. They are given below –

- List
- Tuple
- Range

List

Lists are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

Creating List

Lists in Python can be created by just placing the sequence inside the square brackets[].

Example

```
# Creating a List
```



```
List=[]
print("Initial blank List: ")
print(List)

# Creating a List with
# the use of a String
List=['GeeksForGeeks']
print("\nList with the use of String: ")
print(List)

# Creating a List with
# the use of multiple values
List=["Geeks", "For", "Geeks"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List=[[ 'Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output

Initial blank List:

```
[]
```

List with the use of String:

```
['GeeksForGeeks']
```

List containing multiple values:

```
Geeks
```

Geeks

Multi-Dimensional List:

```
[['Geeks', 'For'], ['Geeks']]
```

Tuple

Tuples are similar to lists, but they can't be modified once they are created. Tuples are commonly used to store data that should not be modified, such as configuration settings or data that is read from a database.

Example

```
# Create a tuple using parentheses
```

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

```
print(t)# Output: (1, 2, 3, 4)
```

```
# Access an item in the tuple using its index
```

```
print(t[1])# Output: 2
```

Output

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

```
2
```

String Data Type

Strings in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

Strings can be used for a variety of actions, including concatenation, slicing, and repetition.

- Concatenation – This process involves connecting two or more threads together.
- Slicing is a method for taking different pieces of string out.
- Repeating a set of instructions, a certain number of times is referred to as repetition.

Creating String

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

Example

```
# 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)
print(type(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)
print(type(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

```
<class 'str'>
```

String with the use of Triple Quotes:

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

```
<class 'str'>
```

Creating a multiline String:

Geeks

For

Life

Python Operators

The operator is a symbol that performs a specific operation between two operands, according to one definition. Operators serve as the foundation upon which logic is constructed in a program in a particular programming language. In every programming language, some operators perform several tasks. Same as other languages, Python also has some operators, and these are given below -

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

Arithmetic Operators

Arithmetic operators used between two operands for a particular operation. There are many arithmetic operators. It includes the exponent (**) operator as well as the + (addition), - (subtraction), * (multiplication), / (divide), % (remainder), and // (floor division) operators.

Operator	Description
+ (Addition)	It is used to add two operands. For example, if $a = 10$, $b = 10 \Rightarrow a+b = 20$
- (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 $a = 20$, $b = 5 \Rightarrow a - b = 15$
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if $a = 20$, $b = 10 \Rightarrow a/b = 2.0$
* (Multiplication)	It is used to multiply one operand with the other. For example, if $a = 20$, $b = 4 \Rightarrow a * b = 80$
% (remainder)	It returns the remainder after dividing the first operand by the second operand. For example, if $a = 20$, $b = 10 \Rightarrow a \% b = 0$
** (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.

Example

1. `a = 32` # Initialize the value of a
2. `b = 6` # Initialize the value of b
3. `print('Addition of two numbers:',a+b)`
4. `print('Subtraction of two numbers:',a-b)`
5. `print('Multiplication of two numbers:',a*b)`
6. `print('Division of two numbers:',a/b)`
7. `print('Reminder of two numbers:',a%b)`
8. `print('Exponent of two numbers:',a**b)`

9. `print('Floor division of two numbers:',a//b)`

Output

Addition of two numbers: 38

Subtraction of two numbers: 26

Multiplication of two numbers: 192

Division of two numbers: 5.333333333333333

Reminder of two numbers: 2

Exponent of two numbers: 1073741824

Floor division of two numbers: 5

Comparison operator

Comparison operators mainly use for comparison purposes. Comparison operators compare the values of the two operands and return a true or false Boolean value in accordance. The example of comparison operators are ==, !=, <=, >=, >, <. In the below table, we explain the works of the 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.

Example

1. `a = 32` # Initialize the value of a
2. `b = 6` # Initialize the value of b
3. `print('Two numbers are equal or not:',a==b)`
4. `print('Two numbers are not equal or not:',a!=b)`
5. `print('a is less than or equal to b:',a<=b)`
6. `print('a is greater than or equal to b:',a>=b)`
7. `print('a is greater b:',a>b)`
8. `print('a is less than b:',a<b)`

Output

Two numbers are equal or not: False

Two numbers are not equal or not: True

a is less than or equal to b: False

a is greater than or equal to b: True

a is greater b: True

a is less than b: False

Assignment Operators

Using the assignment operators, the right expression's value is assigned to the left operand. There are some examples of assignment operators like `=`, `+=`, `-=`, `*=`, `%=`, `**=`, `//=`. In the below table, we explain the works of the operators.

Operator	Description
<code>=</code>	It assigns the value of the right expression to the left operand.
<code>+=</code>	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 <code>a = 10</code> , <code>b = 20</code> \Rightarrow <code>a += b</code> will be equal to <code>a = a + b</code> and therefore, <code>a = 30</code> .

-=	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 remainder 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. The examples of Bitwise operators are bitwise OR ($|$), bitwise AND ($\&$), bitwise XOR (\wedge), negation (\sim), Left shift (\ll), and Right shift (\gg). Consider the case below.

Rules

1. **if** $a = 7$
2. $b = 6$
3. then, binary $(a) = 0111$
4. binary $(b) = 0110$
5. hence, $a \& b = 0011$
6. $a | b = 0111$
7. $a \wedge b = 0100$
8. $\sim a = 1000$

9. Let, Binary of $x = 0101$
10. Binary of $y = 1000$
11. Bitwise OR = 1101
12. 8 4 2 1
13. $1\ 1\ 0\ 1 = 8 + 4 + 1 = 13$
14. Bitwise AND = 0000
15. $0000 = 0$
16. Bitwise XOR = 1101
17. 8 4 2 1
18. $1\ 1\ 0\ 1 = 8 + 4 + 1 = 13$
19. Negation of $x = \sim x = (-x) - 1 = (-5) - 1 = -6$
20. $\sim x = -6$

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.

Example

1. $a = 5$ # initialize the value of a
2. $b = 6$ # initialize the value of b

3. `print('a&b:', a&b)`
4. `print('a|b:', a|b)`
5. `print('a^b:', a^b)`
6. `print('~a:', ~a)`
7. `print('a<<b:', a<<b)`
8. `print('a>>b:', a>>b)`

Output

a&b: 4

a|b: 7

a^b: 3

~a: -6

a<>b: 0

Logical Operators

The assessment of expressions to make decisions typically uses logical operators. The examples of logical operators are and, or, and not. In the case of logical AND, if the first one is 0, it does not depend upon the second one. In the case of logical OR, if the first one is 1, it does not depend on the second one. Python supports the following logical operators.

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.

Example

1. `a = 5` `# initialize the value of a`
2. `print('Is this statement true?:', a > 3 and a < 5)`
3. `print('Any one statement is true?:', a > 3 or a < 5)`
4. `print('Each statement is true then return False and vice-versa:', (not(a > 3 and a < 5)))`

Output

Is this statement true?: False

Any one statement is true?: True

Each statement is true then return False and vice-versa: True

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).

Example

1. `x = ["Rose", "Lotus"]`
2. `print(' Is value Present?', "Rose" in x)`
3. `print(' Is value not Present?', "Riya" not in x)`

Output

Is value Present? True

Is value not Present? True

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.
```

Example

1. `a = ["Rose", "Lotus"]`
2. `b = ["Rose", "Lotus"]`
3. `c = a`
4. `print(a is c)`
5. `print(a is not c)`
6. `print(a is b)`
7. `print(a is not b)`
8. `print(a == b)`
9. `print(a != b)`

output

True

False

False

True

True

False

Expression

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.

Example

```
# Arithmetic Expression
```

```
1+2
```

```
# Comparison Expression
```

```
1<2
```

```
# String Concatenation Expression
```

```
"Hello"+" "+"World"
```

```
# Function Call Expression
```

```
len("Hello")
```

```
# List Indexing Expression
```

```
[1,2,3][1]
```

```
# Tuple Packing Expression
```

```
1,2,3
```

```
# Dictionary Lookup Expression
```

```
{"key":"value"}["key"]
```

Statements

In Python, statements are instructions that perform some sort of action, but do not produce any value. Statements are used to control the flow of the program and carry out various tasks, such as defining variables, making decisions, and looping through data. Some common examples of Python statements include assignment statements, control flow statements (such as if/else, for, and while loops), and import statements.

1. Assignment Statement: `x = 1` assigns the value of 1 to the variable `x`.
2. Control Flow Statement: `if x > 0: print("x is positive")` checks if the value of `x` is positive and prints a message if it is.
3. Import Statement: `import math` imports the `math` module and makes its functions and constants available for use in the program.
4. For Loop Statement: `for i in range(5): print(i)` iterates over the range 0 to 4 and prints each number.

5. While Loop Statement: `while x < 5: x += 1` repeatedly increments the value of x until it is no longer less than 5.
6. Function Definition Statement: `def add(a, b): return a + b` defines a function that takes two arguments and returns their sum

Example

```
# Assignment Statement
```

```
x = 1
```

```
# Control Flow Statement
```

```
if x > 0:
```

```
    print("x is positive")
```

```
# Import Statement
```

```
import math
```

```
# For Loop Statement
```

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

```
    print(i)
```

```
# While Loop Statement
```

```
while x < 5:
```

```
    x += 1
```

```
# Function Definition Statement
```

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

```
    return a + b
```

String operations in Python

Strings are fundamental and essential data structures that every Python programmer works with. In Python, a string is a sequence of characters enclosed within either single quotes ('...') or double quotes ("..."). It is an immutable built-in data structure, meaning once a string is created, it cannot be modified.

1. String Padding: Add Extra Character Elegantly

String padding is a term to adding characters to the beginning or end of a string until it reaches a certain length. It can be useful in formatting text to align with other data or to make it easier to read.

In Python, you can pad a string using the `str.ljust()`, `str.rjust()`, and `str.center()` methods.

Example

1. `text = "Python"`
2. `padded_text = text.rjust(10, '-')`
3. `print(padded_text)`

Output

```
----Python
```

In this example, the `rjust()` method adds dashes to the beginning of the string until it is ten characters long.

2. String Splitting

String splitting refers to dividing a string into multiple substrings based on a specified delimiter or separator. In Python, you can split a string using the `str.split()` method.

Example

1. `text = "Hello world, how are you today?"`
2. `words = text.split()`
3. `print(words)`

Output

```
['Hello', 'world,', 'how', 'are', 'you', 'today?']
```

In this example, the `split ()` method returns a list of substrings, where each substring corresponds to a word in the original string.

3. Concatenate Strings

In Python, we can concatenate strings using the + operator.

Example

1. `string1 = "Hello"`
2. `string2 = "world"`
3. `result = string1 + " " + string2`
4. `print(result)`

Output

Hello world

4. Search for Substring Effective

Finding search string is a common requirement in daily programming. Python comes with the two methods. One is `find()` method ,

Example

1. `title = 'How to search substrings of Python strings'`
2. `print(title.find('string'))`
3. `print(title.find('string'))`
4. `print(title.find('Yang'))`

Output

17

35

5. Reverse the String

Generally, we use the loop to reverse the given string; it can be also reversed using the slicing.

Example

1. `name = "Peter"`
2. `print(name[::-1])`

Output

reteP

6.Deleting the String

As we know that strings are immutable. We cannot delete or remove the characters from the string. But we can delete the entire string using the **del** keyword.

Example

1. `str1 = "JAVATPOINT"`
2. `del str1`
3. `print(str1)`

Output

NameError: name 'str1' is not defined

Boolean expressions

Booleans represent one of two values: **True** or **False**.

Example

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Output

True

False

False

a= 200

b= 33

Example

```
if b > a:  
    print("b is greater than a")  
else:  
    print("b is not greater than a")
```

Output

a is not greater than a

Control Statements in Python

Control statements in Python are a powerful tool for managing the flow of execution. 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.

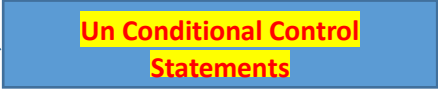
By using control statements effectively, developers can write more efficient and effective code.

A. Conditional Control Construct(Selection, Iteration)

B. Un- Conditional Control Construct (pass, break, continue, exit(), quit())

Python have following types of control statements

1. **Selection** (branching) Statement
2. **Iteration** (looping) Statement
3. **Jumping** (break / continue)Statement



if Statements

The if statement is arguably the most used statement to control loops. For instance:

Code

```
# Python program to show how if statements control loops
```

```
n = 5
for i in range(n):
    if i < 2:
        i += 1
    if i > 2:
        i -= 2
    print(i)
```

Output:

```
1
2
2
1
2
```

Python if - else statements

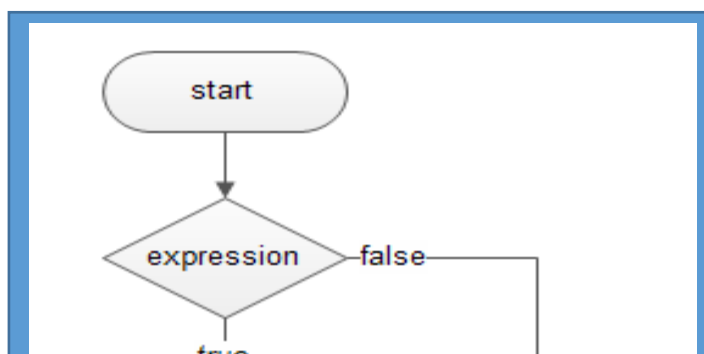
This construct of python program consist of **one if condition with two blocks**. When **condition** becomes true then executes the block given below it. If condition evaluates result as false, it will executes the block given below else.

Syntax:

if (condition):

.....
else:
.....

Flowchart



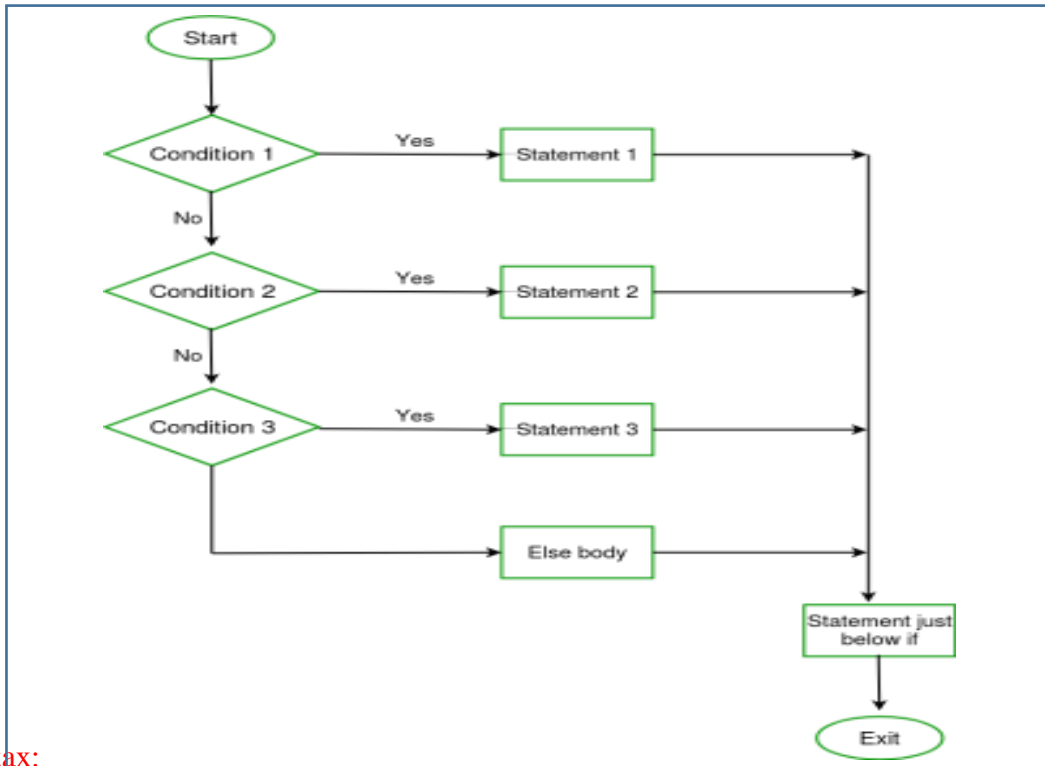
Example-2:

```
N=int(input("Enter Number: "))  
if(n%2==0):  
    print(N," is Even Number")  
  
Else:  
    print(N," is Odd Number")
```

Python Ladder if else statements (if-elif-else)

This construct of python program consist of **more than one if condition**. When first condition evaluates result as true then executes the block given below it. If condition evaluates result as false, it transfer the control at else part to test another condition. So, it is **multi-decision making construct**.

Flowchart



Syntax:

```
if ( condition-1):  
    .....  
    .....  
.elif (condition-2):  
    .....  
    .....
```

```
.elif (condition-3):
```

```
.....  
.....
```

```
else:
```

```
.....  
.....
```

Example

```
var = 100  
if var == 200:  
    print "1 - Got a true expression value"  
    print var  
elif var == 150:  
    print "2 - Got a true expression value"  
    print var  
elif var == 100:  
    print "3 - Got a true expression value"  
    print var  
else:  
    print "4 - Got a false expression value"  
    print var  
  
print "Good bye!"
```

Output

```
3 - Got a true expression value  
100  
Good bye!
```

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

```
# Python program to show how to control the flow of loops with the break statement
```

```
Details = [[19, 'Itika', 'Jaipur'], [16, 'Aman', 'Bihar']]
```

```
for candidate in Details:
```

```
    age = candidate[0]
```

```
    if age <= 18:
```

break

```
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

Code

1. # Python program to show how to control the flow of a loop using a continue statement
2. # Printing only the letters of the string
3. **for** l **in** 'I am a coder':
4. **if** l == ' ':
5. **continue**
6. **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.

Python While Loop

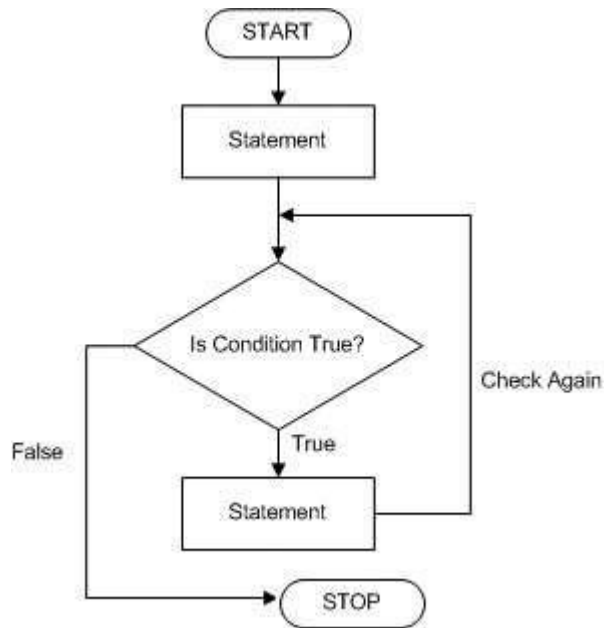
Until a specified criterion is true, a block of statements will be continuously executed in a Python while loop. And the line in the program that follows the loop is run when the condition changes to false.

Syntax of Python While

while expression:

statement(s)

Flowchart



Example

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

Output:

Hello Geek

Hello Geek

Hello Geek

Python for loop

A for loop is used for iterating over a sequence (that is either a list, a tuple, a string etc.) With for loop we can execute a set of statements, and for loop can also execute once for each element in a list, tuple, set etc.

Syntax

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

```
# 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

Input from keyboard

Taking input is a way of interact with users, or get data to provide some result. Python provides two [built-in](#) methods to read the data from the keyboard. These methods are given below.

- `input(prompt)`
- `raw_input(prompt)`

`input()`

The input function is used in all latest version of the Python. It takes the input from the user and then evaluates the expression. The [Python](#) interpreter automatically identifies the whether a user input a string, a number, or a list. Let's understand the following example.

Example -

1. `# Python program showing`
2. `# a use of input()`
- 3.
4. `name = input("Enter your name: ")`
5. `print(name)`

Output:

```
Enter your name: Devansh
Devansh
```

Example - 2

1. `# Python program showing`

2. `# a use of input()`
3. `name = input("Enter your name: ") # String Input`
4. `age = int(input("Enter your age: ")) # Integer Input`
5. `marks = float(input("Enter your marks: ")) # Float Input`
6. `print("The name is:", name)`
7. `print("The age is:", age)`
8. `print("The marks is:", marks)`

Output:

```
Enter your name: Johnson
Enter your age: 21
Enter your marks: 89
The name is: Johnson
The age is 21
The marks is: 89.0
```

raw_input()

The `raw_input` function is used in Python's older version like Python 2.x. It takes the input from the keyboard and return as a string. The Python 2.x doesn't use much in the industry. Let's understand the following example.

Example -

```
# Python program showing
# a use of raw_input()
```

```
name = raw_input("Enter your name : ")
print name
```

Output:

```
Enter your name: Peter
Peter
```

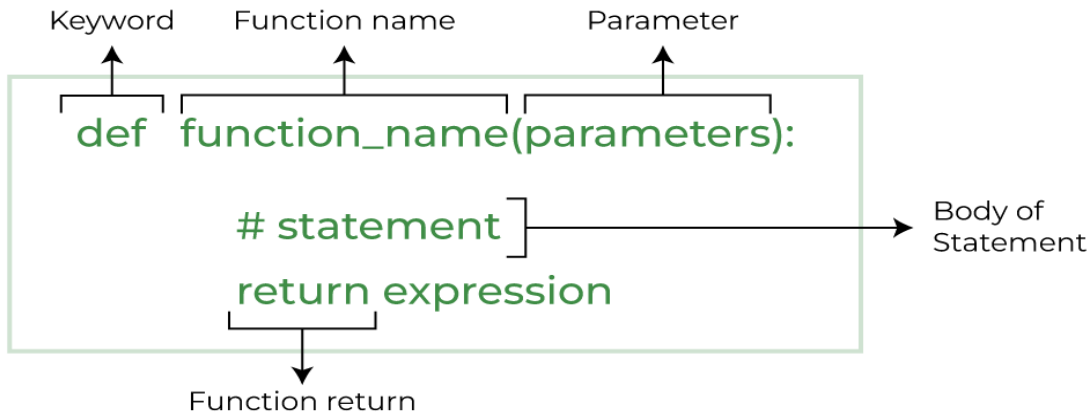
Python Functions

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.

Benefits of Using Functions

- Increase Code Readability
- Increase Code Reusability

The syntax to declare a function is:



Creating a Function

In Python a function is defined using the **def** keyword:

Example

```
def my_function():
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
    print("Hello from a function")
```

```
my_function()
```

Output

Hello from a function

Python Built in Functions

Built-in functions are ones for which the compiler generates inline code at compile time. Every call to a built-in function eliminates a runtime call to the function having the same name in the dynamic library.

Function	Description
<u>abs()</u>	Returns the absolute value of a number
<u>all()</u>	Returns True if all items in an iterable object are true
<u>any()</u>	Returns True if any item in an iterable object is true
<u>ascii()</u>	Returns a readable version of an object. Replaces none-ascii characters with escape character
<u>bin()</u>	Returns the binary version of a number
<u>bool()</u>	Returns the boolean value of the specified object
<u>bytearray()</u>	Returns an array of bytes
<u>bytes()</u>	Returns a bytes object
<u>callable()</u>	Returns True if the specified object is callable, otherwise False
<u>chr()</u>	Returns a character from the specified Unicode code.
<code>classmethod()</code>	Converts a method into a class method

[compile\(\)](#) Returns the specified source as an object, ready to be executed

[complex\(\)](#) Returns a complex number

[delattr\(\)](#) Deletes the specified attribute (property or method) from the specified object

[dict\(\)](#) Returns a dictionary (Array)

[dir\(\)](#) Returns a list of the specified object's properties and methods

[divmod\(\)](#) Returns the quotient and the remainder when argument1 is divided by argument2

[enumerate\(\)](#) Takes a collection (e.g. a tuple) and returns it as an enumerate object

[eval\(\)](#) Evaluates and executes an expression

[exec\(\)](#) Executes the specified code (or object)

[filter\(\)](#) Use a filter function to exclude items in an iterable object

[float\(\)](#) Returns a floating point number

[format\(\)](#) Formats a specified value

[frozenset\(\)](#) Returns a frozenset object

[getattr\(\)](#) Returns the value of the specified attribute (property or method)

[globals\(\)](#) Returns the current global symbol table as a dictionary

[hasattr\(\)](#) Returns True if the specified object has the specified attribute (property/method)

[hash\(\)](#) Returns the hash value of a specified object

[help\(\)](#) Executes the built-in help system

[hex\(\)](#) Converts a number into a hexadecimal value

[id\(\)](#) Returns the id of an object

[input\(\)](#) Allowing user input

[int\(\)](#) Returns an integer number

[isinstance\(\)](#) Returns True if a specified object is an instance of a specified object

[issubclass\(\)](#) Returns True if a specified class is a subclass of a specified object

[iter\(\)](#) Returns an iterator object

[len\(\)](#) Returns the length of an object

[list\(\)](#) Returns a list

[locals\(\)](#) Returns an updated dictionary of the current local symbol table

[map\(\)](#) Returns the specified iterator with the specified function applied to each item

[max\(\)](#) Returns the largest item in an iterable

[memoryview\(\)](#) Returns a memory view object

[min\(\)](#) Returns the smallest item in an iterable

[next\(\)](#) Returns the next item in an iterable

[object\(\)](#) Returns a new object

[oct\(\)](#) Converts a number into an octal

[open\(\)](#) Opens a file and returns a file object

[ord\(\)](#) Convert an integer representing the Unicode of the specified character

[pow\(\)](#) Returns the value of x to the power of y

[print\(\)](#) Prints to the standard output device

[property\(\)](#) Gets, sets, deletes a property

[range\(\)](#) Returns a sequence of numbers, starting from 0 and increments by 1 (by default)

[repr\(\)](#) Returns a readable version of an object

[reversed\(\)](#) Returns a reversed iterator

[round\(\)](#) Rounds a numbers

[set\(\)](#) Returns a new set object

[setattr\(\)](#) Sets an attribute (property/method) of an object

[slice\(\)](#) Returns a slice object

[sorted\(\)](#) Returns a sorted list

<code>staticmethod()</code>	Converts a method into a static method
<code>str()</code>	Returns a string object
<code>sum()</code>	Sums the items of an iterator
<code>super()</code>	Returns an object that represents the parent class
<code>tuple()</code>	Returns a tuple
<code>type()</code>	Returns the type of an object
<code>vars()</code>	Returns the <code>__dict__</code> property of an object
<code>zip()</code>	Returns an iterator, from two or more iterators

user-defined functions in Python

Functions that we define ourselves to do certain specific task are referred as user-defined functions.

Advantages

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmers working on large project can divide the workload by making different functions.

Syntax:

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

Example

```
# Declaring a function  
def fun():  
    print("Inside function")  
  
# Driver's code  
# Calling function  
fun()
```

Output:

Inside function

Parameterized Function

The function may take arguments(s) also called parameters as input within the opening and closing parentheses, just after the function name followed by a colon.

Syntax:

```
def function_name(argument1, argument2, ...):  
    statements  
    .  
    .
```

Example:

```
# Python program to  
  
# demonstrate functions
```

```
# A simple Python function to check
# whether x is even or odd

def evenOdd( x ):

    if (x % 2 == 0):

        print("even")

    else:

        print("odd")

# Driver code

evenOdd(2)

evenOdd(3)
```

Output:

```
even
odd
```

[type conversion](#)

In programming, type conversion is the process of converting data of one type to another. For example: converting int data to str.

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion
- Explicit Conversion - manual type conversion

Implicit Type Conversion

- In certain situations, Python automatically converts one data type to another. This is known as implicit type conversion. The Programming language automatically changes one data type to another in implicit shift of data types.

1. `x = 20`
2. `print("x type:",type(x))`
3. `y = 0.6`
4. `print("y type:",type(y))`
5. `a = x + y`
6. `print(a)`
7. `print("z type:",type(z))`

Output:

```
x type: <class 'int'>
y type: <class 'float' >20.6
a type: <class 'float'>
```

Explicit Type Conversion

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. With explicit type conversion, there is a risk of data loss since we are forcing an expression to be changed in some specific data type. Various forms of explicit type conversion are explained below:

Example

```
# Python code to demonstrate Type conversion

# using int(), float()

# initializing string
s = "10010"

# printing string converting to int base 2
c = int(s,2)

print ("After converting to integer base 2 : ", end="")

print (c)

# printing string converting to float
```

```
e = float(s)

print ("After converting to float : ", end="")

print (e)
```

Output:

After converting to integer base 2 : 18

After converting to float : 10010.0

Type Coercion

There are occasions when we would like to convert a variable from one data type to another. This is referred to as **type coercion**. We can coerce a variable to another data type by passing it to a function whose name is identical to the desired data type. For instance, if we want to convert a variable **x** to an integer, we would use the command **int(x)**. If we want to convert a variable **y** to a float, we would use **float(y)**.

Example

```
# Coercing an int to a float.
```

```
x_int = 19
x_float = float(x_int)
```

```
print(x_float)
print(type(x_float))
```

Output

```
19.0
<class 'float'>
```

```
# Coercing a float to an int.
```

```
y_float = 6.8
y_int = int(y_float)
```

```
print(y_int)
print(type(y_int))
```

Output

```
6
```

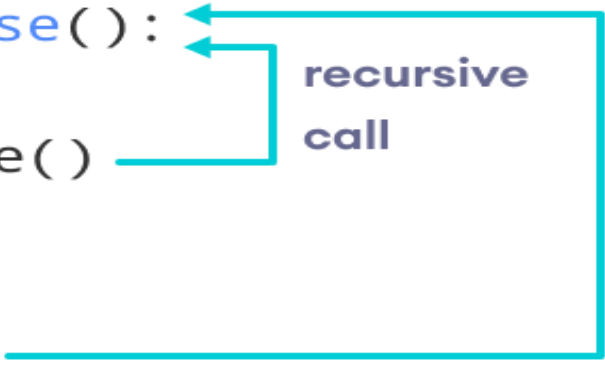
<class 'int'>

Python Recursion

Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

```
def recurse():  
    ...  
    recurse()  
    ...  
  
recurse()
```



Example

```
# Factorial of a number using recursion  
  
def recur_factorial(n):  
    if n == 1:  
        return n  
    else:  
        return n*recur_factorial(n-1)  
  
num = 7  
  
# check if the number is negative  
if num < 0:  
    print("Sorry, factorial does not exist for negative numbers")  
elif num == 0:  
    print("The factorial of 0 is 1")  
else:  
    print("The factorial of", num, "is", recur_factorial(num))
```

Output

The factorial of 7 is 5040

Unit II

Strings

String can be defined as a **sequence of characters**.

Strings can be considered as a special type of sequence, where all its elements are characters. For example, string "Hello, World" is basically a sequence ['H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd'] and its length can be calculated by counting number of characters inside the sequence, which is 12.

Declaration of Strings

```
>>> mystring = "This is not my first String"

>>> print (mystring);

This is not my first String
```

Example

```
str = 'Hello World!'

print str # Prints complete string

print str[0] # Prints first character of the string

print str[2:5] # Prints characters starting from 3rd to 5th

print str[2:] # Prints string starting from 3rd character

print str * 2 # Prints string two times

print str + "TEST" # Prints concatenated string
```

Output

This will produce the following result –

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

len() function

The function `len()` is one of Python's built-in functions. It returns the length of an object. For example, it can return the number of items in a list.

Syntax: `len(object)`

Here object- Required. An object. Must be a sequence or a collection

Example

```
mylist = ["apple", "banana", "cherry"]
```

```
x = len(mylist)
```

Output

3

Example

```
greeting = "Good Day!"
```

```
len(greeting)
```

Output

9

Example

```
# Python program to demonstrate the use of
```

```
# len() method
```

```
# with tuple
```



```
tup = (1,2,3)

print(len(tup))

# with list

l = [1,2,3,4]

print(len(l))
```

Output:

3

4

Python slice()

can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

Example

```
b = "Hello, World!"
print(b[2:5])
```

Output

llo

The `slice()` function returns a slice object that is used to slice any sequence (string, tuple, list, range, or bytes).

Example

```
text = 'Python Programing'
```

```
# get slice object to slice Python
sliced_text = slice(6)

print(text[sliced_text])
```

```
# Output: Python
```

slice() Syntax

The syntax of slice() is:

```
slice(start, stop, step)
```

slice() Parameters

slice() can take three parameters:

- **start (optional)** - Starting integer where the slicing of the object starts. Default to None if not provided.
- **stop** - Integer until which the slicing takes place. The slicing stops at index **stop -1 (last element)**.
- **step (optional)** - Integer value which determines the increment between each index for slicing. Defaults to None if not provided.

Example 1: Create a slice object for slicing

```
# contains indices (0, 1, 2)
result1 = slice(3)
print(result1)

# contains indices (1, 3)
result2 = slice(1, 5, 2)
```

```
print(slice(1, 5, 2))
```

[Run Code](#)

Output

```
slice(None, 3, None)
```

```
slice(1, 5, 2)
```

Example 2: Get substring using slice object

```
# Program to get a substring from the given string
```

```
py_string = 'Python'
```

```
# stop = 3
```

```
# contains 0, 1 and 2 indices
```

```
slice_object = slice(3)
```

```
print(py_string[slice_object]) # Pyt
```

```
# start = 1, stop = 6, step = 2
```

```
# contains 1, 3 and 5 indices
```

```
slice_object = slice(1, 6, 2)
```

```
print(py_string[slice_object]) # yhn
```

[Run Code](#)

Output

```
Pyt
```

```
yhn
```

String Traversal

Traversing a string. Traversing just means to process every character in a string, usually from left end to right end. Python allows for 2 ways to do this – both useful but not identical.

Iterate over a list in Python

Method 1: Using For loop.

Method 2: For loop and range()

Method 3: Using a while loop.

Method 4: Using list comprehension.

Method 5: Using enumerate()

Method 6: Using NumPy.

Method 7: Using the iter function and the next function.

Method 8: Using the map() function.

Method 1: Using For loop

We can iterate over a list in Python by using a simple [For loop](#).

```
# Python3 code to iterate over a list
```

```
list = [1, 3, 5, 7, 9]
```

```
# Using for loop
```

```
for i in list:
```

```
    print(i)
```

Output:

1

3

5

7

9

Method 2: For loop and range()

In case we want to use the traditional for loop which iterates from number x to number y.

- Python3

```
# Python3 code to iterate over a list
```

```
list = [1, 3, 5, 7, 9]
```

```
# getting length of list
```

```
length = len(list)
```

```
# Iterating the index
```

```
# same as 'for i in range(len(list))'
```

```
for i in range(length):
```

```
    print(list[i])
```

Output:

1

3

5

7

9

Method 3: Using a while loop

We can also iterate over a Python list using a [while loop](#).

- Python3

```
# Python3 code to iterate over a list
```

```
list = [1, 3, 5, 7, 9]
```

```
# Getting length of list
```

```
length = len(list)
```

```
i = 0
```

```
# Iterating using while loop
```

```
while i < length:
```

```
    print(list[i])
```

```
    i += 1
```

Output:

1

3

5

7

9

Method 4: Using list comprehension

We can use [list comprehension](#) (possibly the most concrete way) to iterate over a list in Python.

- Python3

```
# Python3 code to iterate over a list
```

```
list = [1, 3, 5, 7, 9]
```

```
# Using list comprehension
```

```
[print(i) for i in list]
```

Output:

1

3

5

7

9

Method 5: Using enumerate()

If we want to convert the list into an iterable list of tuples (or get the index based on a condition check, for example in linear search you might need to save the index of minimum element), you can use the [enumerate\(\) function](#).

- Python3

```
# Python3 code to iterate over a list
```

```
list = [1, 3, 5, 7, 9]
```

```
# Using enumerate()
```

```
for i, val in enumerate(list):
```

```
print (i, ",",val)
```

Output:

0 , 1

1 , 3

2 , 5

3 , 7

4 , 9

Method 6: Using NumPy

For very large n-dimensional lists (for example an image array), it is sometimes better to use an external library such as [numpy](#).

- Python3

```
# Python program for
```

```
# iterating over array
```

```
import numpy as geek
```

```
# creating an array using
```

```
# arrange method
```

```
a = geek.arange(9)
```

```
# shape array with 3 rows
```



```
# and 4 columns

a = a.reshape(3, 3)

# iterating an array

for x in geek.nditer(a):

    print(x)
```

Output:

```
0
1
2
3
4
5
6
7
8
```

Method 7: Using the iter function and the next function

Here is an additional approach using the iter function and the next function:

- Python3

```
# Python3 code to iterate over a list

list = [1, 3, 5, 7, 9]
```

```
# Create an iterator object using the iter function

iterator = iter(list)

# Use the next function to retrieve the elements of the iterator

try:

    while True:

        element = next(iterator)

        print(element)

except StopIteration:

    pass
```

Output

1
3
5
7
9

Method 8: Using the map() function

Use the [map\(\)](#) function to apply a function to each element of a list.

- Python3

```
# Define a function to print each element
```

```
def print_element(element):

    print(element)

# Create a list

my_list = [1, 3, 5, 7, 9]

# Use map() to apply the print_element() function to each element of the list

result = map(print_element, my_list)

# Since map() returns an iterator, we need to consume

# the iterator in order to see the output

for _ in result:

    pass
```

Output

1
3
5
7
9

Python Escape Characters

As the name suggests, an escape sequence is a sequence of characters with special meaning when used inside a string or a character. If an escape sequence is designated to a Non-Printable Character or a Control Code, then the sequence is called a control character.

Escape Character	Meaning
-------------------------	----------------

Syntax: The characters need to be preceded by a backslash character

Example: \n, \t etc.

If an escape sequence is designated to a Non-Printable Character or a Control Code, then the sequence is called a control character.

<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	backslash
<code>\n</code>	New line
<code>\r</code>	Carriage Return
<code>\t</code>	Horizontal tab
<code>\b</code>	Backspace
<code>\f</code>	form feed
<code>\v</code>	vertical tab
<code>\0</code>	Null character
<code>\N{name}</code>	Unicode Character Database named Lookup
<code>\uxxxxxxxx</code>	Unicode Character with 16-bit hex value XXXX

<pre>\Uxxxxxxxx</pre>	<p>Unicode Character with 32-bit hex value XXXXXXXXXX</p>
<pre>\ooo</pre>	<p>Character with octal value OOO</p>
<pre>\xhh</pre>	<p>Character with hex value HH</p>
<p>Example</p> <pre># A string with a recognized escape sequence print("I will go\tHome") # A string with a unrecognized escape sequence print("See you\jtomorrow")</pre> <p>Output</p> <pre>I will go Home See you\jtomorrow</pre> <p>Example</p>	

```
# sample string
```

```
s = "I love to use \t instead of using 4 spaces"
```

```
# normal output
```

```
print(s)
```

```
# doubling line backslashes
```

```
s = "I love to use \\t instead of using 4 spaces"
```

```
# output after doubling
```

```
print(s)
```

Output:

I love to use instead of using 4 spaces

I love to use \t instead of using 4 spaces

String formatting operator

String formatting in Python allows you to create dynamic strings by combining variables and values.

Example

```
#!/usr/bin/python
```

```
print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

output

My name is Zara and weight is 21 kg!

There are five different ways to perform string formatting in Python

- Formatting with % Operator.
- Formatting with format() string method.
- Formatting with string literals, called f-strings.
- Formatting with String Template Class
- Formatting with center() string method.

Formatting string using % Operator

```
print("The mangy, scrawny stray dog %s gobbled down" % 'hurriedly' +  
      "the grain-free, organic dog food.")
```

Output:

The mangy, scrawny stray dog hurriedly gobbled down the grain-free, organic dog food.

Formatting using format() Method

[Format\(\) method](#) was introduced with Python3 for handling complex string formatting more efficiently.

Syntax: 'String here {} then also {}'.format('something1', 'something2')

Formatting String using format() Method

This code is using {} as a placeholder and then we have called.format() method on the 'equal' to the placeholder.

Python3

```
print('We all are {}'.format('equal'))
```

Output:

We all are equal.

Python f-string

To create an f-string in Python, prefix the string with the letter “f”. The string itself can be formatted in much the same way that you would with **str. format()**. F-strings provide a concise and convenient way to embed Python expressions inside string literals for formatting.

String Formatting with F-Strings

In this code, the f-string **f”My name is {name}.”** is used to interpolate the value of the name variable into the string.

Example

```
name = 'Ele'

print(f"My name is {name}.")
```

Output:

My name is Ele.

Python String Template Class

Template Class allows us to create simplified syntax for output specification. The format uses placeholder names formed by \$ with valid Python [identifiers](#) (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing \$\$ creates a single escaped \$:

Example

```
# Python program to demonstrate  
  
# string interpolation  
  
from string import Template  
  
n1 = 'Hello'  
  
n2 = 'GeeksforGeeks'  
  
# made a template which we used to  
  
# pass two variable so n3 and n4  
  
# formal and n1 and n2 actual  
  
n = Template('$n3 ! This is $n4.')  
  
# and pass the parameters into the
```

```
# template string.  
  
print(n.substitute(n3=n1, n4=n2))
```

Output

Hello ! This is GeeksforGeeks.

Python String center() Method

The center() method is a built-in method in [Python](#)'s str class that returns a new string that is centered within a string of a specified width.

Example

```
string = "GeeksForGeeks!"  
  
width = 30  
  
centered_string = string.center(width)  
  
print(centered_string)
```

Output :

GeeksForGeeks!

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.

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 an empty Tuple
```

```
Tuple1 = ()
```

```
print("Initial empty Tuple: ")
```

```
print(Tuple1)
```

```
# Creating a Tuple
```

```
# with the use of string
```

```
Tuple1 = ('Geeks', 'For')
```

```
print("\nTuple with the use of String: ")
```

```
print(Tuple1)
```

```
# Creating a Tuple with
```

```
# the use of list
```

```
list1 = [1, 2, 4, 5, 6]

print("\nTuple using List: ")

print(tuple(list1))

# Creating a Tuple

# with the use of built-in function

Tuple1 = tuple('Geeks')

print("\nTuple with the use of function: ")

print(Tuple1)
```

Output:

Initial empty Tuple:

()

Tuple with the use of String:

('Geeks', 'For')

Tuple using List:

(1, 2, 4, 5, 6)

Tuple with the use of function:

('G', 'e', 'e', 'k', 's')

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.

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 an empty Tuple

Tuple1 = ()

print("Initial empty Tuple: ")

print(Tuple1)

# Creating a Tuple

# with the use of string

Tuple1 = ('Geeks', 'For')

print("\nTuple with the use of String: ")

print(Tuple1)
```

```
# Creating a Tuple with  
  
# the use of list  
  
list1 = [1, 2, 4, 5, 6]  
  
print("\nTuple using List: ")  
  
print(tuple(list1))  
  
# Creating a Tuple  
  
# with the use of built-in function  
  
Tuple1 = tuple('Geeks')  
  
print("\nTuple with the use of function: ")  
  
print(Tuple1)
```

Output:

Initial empty Tuple:

()

Tuple with the use of String:

('Geeks', 'For')

Tuple using List:

(1, 2, 4, 5, 6)

Tuple with the use of function:

```
('G', 'e', 'e', 'k', 's')
```

Accessing values in 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).

```
# Accessing Tuple

# with Indexing

Tuple1 = tuple("Geeks")

print("\nFirst element of Tuple: ")

print(Tuple1[0])

# Tuple unpacking

Tuple1 = ("Geeks", "For", "Geeks")

# This line unpack

# values of Tuple1
```



```
a, b, c = Tuple1

print("\nValues after unpacking: ")

print(a)

print(b)

print(c)
```

Output:

First element of Tuple:

G

Values after unpacking:

Geeks

For

Geeks

Tuple assignment

- v An assignment to all of the elements in a tuple using a single assignment statement.
- v Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.
- v The left side is a tuple of variables; the right side is a tuple of values.
- v Each value is assigned to its respective variable.

Example

```
fruits = ("apple", "banana", "cherry")
```

Output

```
("apple", "banana", "cherry")
```

Example

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")
```

```
(green, yellow, red) = fruits
```

```
print(green)
```

```
print(yellow)
```

```
print(red)
```

Output

Apple

Banana

Cherry

Tuples as return values

Functions can return tuples as return values.

Example

```
def circleInfo(r):
```

```
    """ Return (circumference, area) of a circle of radius r """
```

```
    c = 2 * 3.14159 * r
```

```
    a = 3.14159 * r * r
```

```
    return c, a
```

```
print(circleInfo(10))
```

Output

62.8318, 314.159

Basic tuple operations

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Python Expression	Results	Description
<code>len((1, 2, 3))</code>	3	Length
<code>(1, 2, 3) + (4, 5, 6)</code>	<code>(1, 2, 3, 4, 5, 6)</code>	Concatenation
<code>('Hi!') * 4</code>	<code>('Hi!', 'Hi!', 'Hi!', 'Hi!')</code>	Repetition
<code>3 in (1, 2, 3)</code>	True	Membership
<code>for x in (1, 2, 3): print x,</code>	1 2 3	Iteration

Built-in tuple functions

Built-in functions are pre-defined in the programming language's library for the programming to directly call the functions wherever required in the program for achieving certain functional operations.

Built-in Function	Description
<u>all()</u>	Returns true if all element are true or if tuple is empty
<u>any()</u>	return true if any element of the tuple is true. if tuple is empty, return false
<u>len()</u>	Returns length of the tuple or size of the tuple
<u>enumerate()</u>	Returns enumerate object of tuple
<u>max()</u>	return maximum element of given tuple
<u>min()</u>	return minimum element of given tuple
<u>sum()</u>	Sums up the numbers in the tuple
<u>sorted()</u>	input elements in the tuple and return a new sorted list
<u>tuple()</u>	Convert an list to a tuple.

Example:

```
>>> T1=(10,20,30,40)
```

```
>>> len(T1)
```

```
4
```

```
#There are 4 element in tuple.
```

Example:

```
>>> T1=[10,20,30,40]
```

```
>>> max(T1)
```

```
40
```

```
# 40 is the maximum value in tuple T1.
```

Example 1:

```
>>> T1=[10,20,30,40]
```

```
>>> min(T1)
```

```
10
```

```
#10 is the minimum value.
```

Example 1:

```
>>> T1=[13,18,11,16,18,14]
>>> T1.count(18) #18 appears twice in tuple T1.
2
```

Example 2 – Creating a tuple from a [list](#)

```
>>>t=tuple([1,2,3])
>>>t
(1,2,3)
```

List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage.

Lists are created using square brackets

Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

output

```
["apple", "banana", "cherry"]
```

Access Items

access the list items by referring to the index number

output

banana

Range of Indexes

can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

Example

Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:5])
```

output

```
"cherry", "orange", "kiwi"
```

Example

This example returns the items from the beginning to "orange":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[:4])
```

output

```
["apple", "banana", "cherry", "orange",]
```

Traversing a list

One of the most common methods to traverse a Python list is to use a for loop, and they are very similar to other programming languages. Alternatively, you can also use the range() method to have more control over your for loop

Traversing Using For Loop and Range Method

Loop Through a List

You can loop through the list items by using a **for** loop:

Example

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
```

```
for x in thislist:
```

```
    print(x)
```

output

apple

banana

cherry

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the **range()** and **len()** functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]
```

```
for i in range(len(thislist)):
```

```
    print(thislist[i])
```

output

apple

banana

cherry

Using a While Loop

You can loop through the list items by using a **while** loop.

Use the **len()** function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Example

Print all items, using a **while** loop to go through all the index numbers

```
thislist = ["apple", "banana", "cherry"]
```

```
i = 0
```

```
while i < len(thislist):
```

```
    print(thislist[i])
```

```
    i = i + 1
```

output

apple

banana

cherry

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

Example

A short hand **for** loop that will print all items in a list:

```
thislist = ["apple", "banana", "cherry"]  
[print(x) for x in thislist]
```

output

apple

banana

cherry

Deleting elements from list

Remove Specified Item

The **remove()** method removes the specified item.

Example

Remove "banana":

```
thislist = ["apple", "banana", "cherry"]  
thislist.remove("banana")  
print(thislist)
```

output

["apple", "cherry"]

Remove Specified Index

The **pop()** method removes the specified index.

Example

Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
```

```
thislist.pop(1)
```

```
print(thislist)
```

output

```
["apple", "cherry"]
```

If you do not specify the index, the `pop()` method removes the last item.

Example

Remove the last item:

```
thislist = ["apple", "banana", "cherry"]
```

```
thislist.pop()
```

```
print(thislist)
```

output

```
["apple", " banana "]
```

The `del` keyword also removes the specified index:

Example

Remove the first item:

```
thislist = ["apple", "banana", "cherry"]
```

```
del thislist[0]
```

```
print(thislist)
```

output

```
[ "banana", "cherry"]
```

The `del` keyword can also delete the list completely.

Example

Delete the entire list:

```
thislist = ["apple", "banana", "cherry"]
```

```
del thislist
```

output

```
NameError: name 'thislist' is not defined
```

Clear the List

The `clear()` method empties the list.

The list still remains, but it has no content.

Example

Clear the list content:

```
thislist = ["apple", "banana", "cherry"]
```

```
thislist.clear()
```

```
print(thislist)
```

output

```
[]
```

Built-in list operators & methods

S.no	Method	Description

1	<u>append()</u>	Used for appending and adding elements to the end of the List.
2	<u>copy()</u>	It returns a shallow copy of a list
3	<u>clear()</u>	This method is used for removing all items from the list.
4	<u>count()</u>	These methods count the elements
5	<u>extend()</u>	Adds each element of the iterable to the end of the List
6	<u>index()</u>	Returns the lowest index where the element appears.
7	<u>insert()</u>	Inserts a given element at a given index in a list.
8	<u>pop()</u>	Removes and returns the last value from the List or the given index value.
9	<u>remove()</u>	Removes a given object from the List.
10	<u>reverse()</u>	Reverses objects of the List in place.
11	<u>sort()</u>	Sort a List in ascending, descending, or user-defined order
12	<u>min()</u>	Calculates the minimum of all the elements of the List
13	<u>max()</u>	Calculates the maximum of all the elements of the List

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

Example

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

print(Dict)
```

Output:

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

Creating a Dictionary

In [Python](#), a dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.

```
# Creating a Dictionary

# with Integer Keys

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

print("\nDictionary with the use of Integer Keys: ")
```

```
print(Dict)

# Creating a Dictionary

# with Mixed keys

Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}

print("\nDictionary with the use of Mixed Keys: ")

print(Dict)
```

Output:

Dictionary with the use of Integer Keys:

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

Dictionary with the use of Mixed Keys:

```
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
```

Accessing elements of a Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets.

Python3

```
# Python program to demonstrate

# accessing a element from a Dictionary
```

```
# Creating a Dictionary
```

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

```
# accessing a element using key
```

```
print("Accessing a element using key:")
```

```
print(Dict['name'])
```

```
# accessing a element using key
```

```
print("Accessing a element using key:")
```

```
print(Dict[1])
```

Output:

Accessing a element using key:

For

Accessing a element using key:

Geeks

Updating dictionary

The `update()` method inserts the specified items to the dictionary.

The specified items can be a dictionary, or an iterable object with key value pairs.

Syntax

```
dictionary.update(iterable)
```


Parameter Values

Parameter	Description
-----------	-------------

iterable

A dictionary or an iterable object with key value pairs, that will be inserted to the

Example

Insert an item to the dictionary:

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
car.update({"color": "White"})  
  
print(car)
```

Output

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}
```

Deleting elements from dictionary

Removing Items from a Dictionary

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.pop("model")
```

output

```
{'brand': 'Ford', 'year': 1964}
```

Example

The `del` keyword removes the item with the specified key name:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict["model"]  
print(thisdict)
```

output

```
{'brand': 'Ford', 'year': 1964}
```

Operations in dictionary

1. Definition operations

These operations allow us to define or create a dictionary.

1.1. { }

Creates an empty dictionary or a dictionary with some initial values.

```
y = {} x = {1: "one", 2: "two", 3: "three"}
```

2. Mutable operations

These operations allow us to work with dictionaries, but altering or modifying their previous definition.

2.1. []

Adds a new pair of key and value to the dictionary, but in case that the key already exists in the dictionary, we can update the value.

```
y = {}
```

Built-in dictionary methods

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary

[copy\(\)](#) Returns a copy of the dictionary

[fromkeys\(\)](#) Returns a dictionary with the specified keys and value

[get\(\)](#) Returns the value of the specified key

[items\(\)](#) Returns a list containing a tuple for each key value pair

[keys\(\)](#) Returns a list containing the dictionary's keys

[pop\(\)](#) Removes the element with the specified key

[popitem\(\)](#) Removes the last inserted key-value pair

[setdefault\(\)](#) Returns the value of the specified key. If the key does not exist: insert the key, with the specified value

[update\(\)](#) Updates the dictionary with the specified key-value pairs

[values\(\)](#) Returns a list of all the values in the dictionary

UNIT III

FILES AND EXCEPTIONS

Introduction to File Input and Output

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a file

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

```
file object = open(<file-name>, <access-mode>, <buffering>)
```

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

SN	Access mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.

10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

The simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example

#opens the file file.txt in read mode

```
fileptr = open("file.txt", "r")
```

if fileptr:

```
    print("file is opened successfully")
```

Output:

```
<class '_io.TextIOWrapper'>
```

```
file is opened successfully
```

In the above code, passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement.

The close() method

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

Syntax

```
fileobject.close()
```

Consider the following example.

```
# opens the file file.txt in read mode
```

```
fileptr = open("file.txt", "r")
```

```
if fileptr:
```

```
    print("file is opened successfully")
```

```
#closes the opened file
```

```
fileptr.close()
```

NOTE

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

```
try:
```

```
    fileptr = open("file.txt")
```

```
    # perform file operations
```

```
finally:
```

```
    fileptr.close()
```

The with statement

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.

```
with open(<file name>, <access mode>) as <file-pointer>:  
#statement suite
```

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Consider the following example.

Example

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

```
content = f.read();  
print(content)
```

Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

Example

```
# open the file.txt in append mode. Create a new file if no such file exists.
```

```
fileptr = open("file2.txt", "w")

# appending the content to the file
fileptr.write("""Python is the modern day language. It makes things so simple.
It is the fastest-growing programing language""")

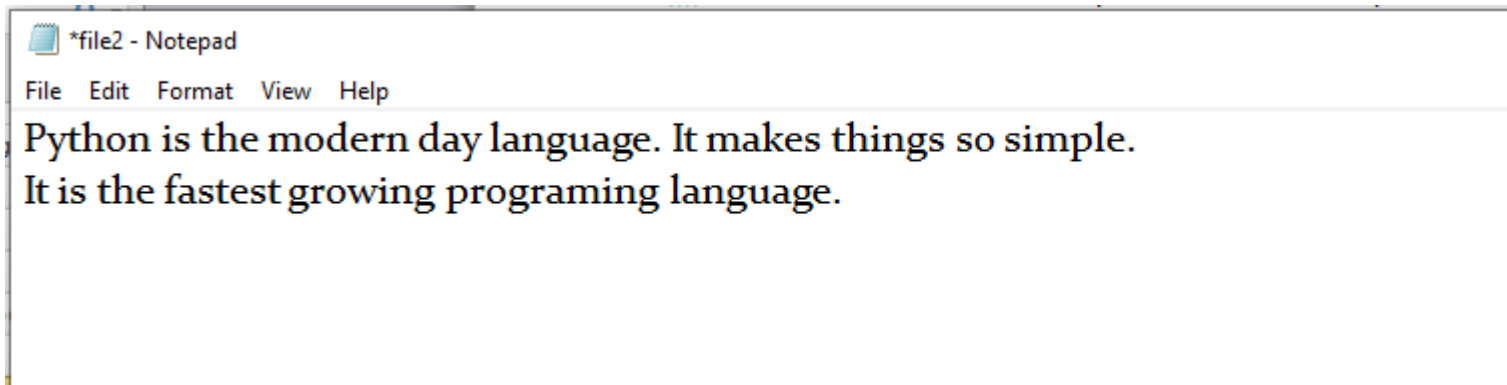
# closing the opened the file
fileptr.close()
```

Output:

File2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

Snapshot of the file2.txt



We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content using the **write()** function.

Example 2

```
#open the file.txt in write mode.
fileptr = open("file2.txt", "a")

#overwriting the content of the file
fileptr.write(" Python has an easy syntax and user-friendly interaction.")
```

#closing the opened file

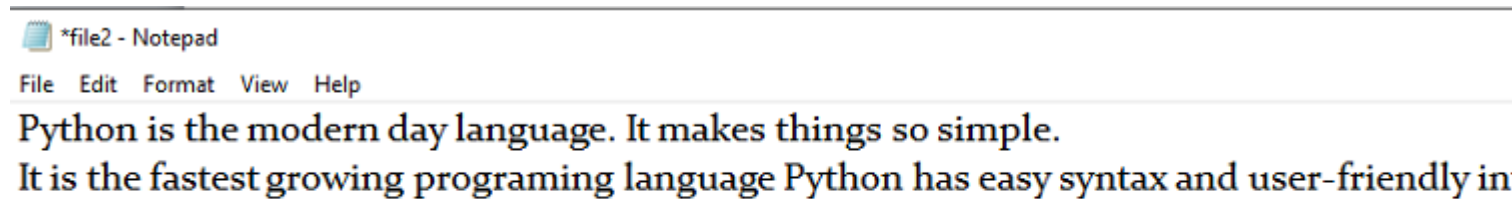
```
fileptr.close()
```

Output:

Python is the modern day language. It makes things so simple.

It is the fastest growing programming language Python has an easy syntax and user-friendly interaction.

Snapshot of the file2.txt



We can see that the content of the file is modified. We have opened the file in **a** mode and it appended to the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

Syntax:

```
fileobj.read(<count>)
```

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is 0, it may read the content of the file until the end.

Consider the following example.

Example

```
#open the file.txt in read mode. causes error if no such file exists.
```

```
fileptr = open("file2.txt", "r")
```

```
#stores all the data of the file into the variable content
```

```
content = fileptr.read(10)
```

```
# prints the type of the data stored in the file
```

```
print(type(content))
```

```
#prints the content of the file
```

```
print(content)
```

```
#closes the opened file
```

```
fileptr.close()
```

Output:

```
<class 'str'>
```

```
Python is
```

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as 10, so it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

```
content = fileptr.read()
```

```
print(content)
```

Output:

```
Python is the modern-day language. It makes things so simple.
```

```
It is the fastest-growing programming language Python has easy an syntax and user-friendly interaction.
```

Read file through for loop

We can read the file using for loop. Consider the following example.

```
#open the file.txt in read mode. causes an error if no such file exists.
```

```
fileptr = open("file2.txt", "r");  
#running a for loop  
for i in fileptr:  
    print(i) # i contains each line of the file
```

Output:

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads text from the beginning, i.e., if we use the **readline()** method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file "**file2.txt**" lines. Consider the following example.

Example 1: Reading lines using **readline()** function

```
#open the file.txt in read mode. causes error if no such file exists.  
fileptr = open("file2.txt", "r");  
#stores all the data of the file into the variable content  
content = fileptr.readline()  
content1 = fileptr.readline()  
#prints the content of the file  
print(content)  
print(content1)  
#closes the opened file  
fileptr.close()
```

Output:

Python is the modern day language.

It makes things so simple.

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the of **file(EOF)** is reached.

Example 2: Reading Lines Using readlines() function

```
#open the file.txt in read mode. causes error if no such file exists.
```

```
fileptr = open("file2.txt","r");
```

```
#stores all the data of the file into the variable content
```

```
content = fileptr.readlines()
```

```
#prints the content of the file
```

```
print(content)
```

```
#closes the opened file
```

```
fileptr.close()
```

Output:

```
['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly int
```

Creating a new file

The new file can be created by using one of the following access modes with the function open().

x: it creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists.

specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example 1

```
#open the file.txt in read mode. causes error if no such file exists.
```

```
fileptr = open("file2.txt","x")
```

```
print(fileptr)
```

```
if fileptr:
```

```
    print("File created successfully")
```

Output:

```
<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>
```

```
File created successfully
```

File Pointer positions

Python provides the `tell()` method which is used to print the byte number at which the file pointer currently is. Consider the following example.

```
# open the file file2.txt in read mode
```

```
fileptr = open("file2.txt","r")
```

```
#initially the filepointer is at 0
```

```
print("The filepointer is at byte :",fileptr.tell())
```

```
#reading the content of the file
```

```
content = fileptr.read();
```

```
#after the read operation file pointer modifies. tell() returns the location of the fileptr.
```

```
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

The filepointer is at byte : 0

After reading, the filepointer is at: 117

Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to access content at various locations.

For this purpose, the Python provides us the `seek()` method which enables us to modify the file pointer position externally.

The syntax to use the `seek()` method is given below.

Syntax:

```
<file-ptr>.seek(offset[, from])
```

The `seek()` method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

Example

```
# open the file file2.txt in read mode
```

```
fileptr = open("file2.txt", "r")
```

```
#initially the filepointer is at 0
```

```
print("The filepointer is at byte :", fileptr.tell())
```

```
#changing the file pointer location to 10.
```



```
fileptr.seek(10);
```

`#tell()` returns the location of the `fileptr`.

```
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

The filepointer is at byte : 0

After reading, the filepointer is at: 10

Renaming the file

The Python `os` module enables interaction with the operating system. The `os` module provides the functions that a processing operations like renaming, deleting, etc. It provides us the `rename()` method to rename the specified file to a new name. The syntax to use the `rename()` method is given below.

Syntax:

```
rename(current-name, new-name)
```

The first argument is the current file name and the second argument is the modified name. We can change the file name by using two arguments.

Example 1:

```
import os
```

```
#rename file2.txt to file3.txt
```

```
os.rename("file2.txt", "file3.txt")
```

Output:

The above code renamed current `file2.txt` to `file3.txt`

Removing the file

The `os` module provides the `remove()` method which is used to remove the specified file. The syntax to use the `remove()` method is given below.

```
remove(file-name)
```

Example 1

```
import os;
#deleting the file named file3.txt
os.remove("file3.txt")
```

Creating the new directory

The `makedirs()` method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

Syntax:

```
makedirs(directory name)
```

Example 1

```
import os
#creating a new directory with the name new
os.makedirs("new")
```

The `getcwd()` method

This method returns the current working directory.

The syntax to use the `getcwd()` method is given below.

Syntax

```
os.getcwd()
```

Example

```
import os  
os.getcwd()
```

Output:

```
'C:\\Users\\DEVANSH SHARMA'
```

Changing the current working directory

The `chdir()` method is used to change the current working directory to a specified directory.

The syntax to use the `chdir()` method is given below.

Syntax

```
chdir("new-directory")
```

Example

```
import os  
# Changing current directory with the new directory  
os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")  
#It will display the current working directory  
os.getcwd()
```

Output:

```
'C:\\Users\\DEVANSH SHARMA\\Documents'
```

Deleting directory

The `rmdir()` method is used to delete the specified directory.

The syntax to use the `rmdir()` method is given below.

Syntax

```
os.rmdir(directory name)
```

Example 1

```
import os
#removing the new directory
os.rmdir("directory_name")
```

It will remove the specified directory.

Writing Python output to the files

In Python, there are the requirements to write the output of a Python script to a file.

The `check_call()` method of module `subprocess` is used to execute a Python script and write the output of that script to a file.

The following example contains two python scripts. The script `file1.py` executes the script `file.py` and writes its output to a file `output.txt`.

Example

file.py

```
temperatures=[10,-20,-289,100]
def c_to_f(c):
    if c < -273.15:
        return "That temperature doesn't make sense!"
    else:
        f=c*9/5+32
        return f
for t in temperatures:
    print(c_to_f(t))
```

file.py

```
import subprocess
```

```
with open("output.txt", "wb") as f:
```

```
    subprocess.check_call(["python", "file.py"], stdout=f)
```

The file related methods

The file object provides the following methods to manipulate the files on various operating systems.

SN	Method	Description
1	file.close()	It closes the opened file. The file once closed, it can't be read or write anymore.
2	File.fush()	It flushes the internal buffer.
3	File.fileno()	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
4	File.isatty()	It returns true if the file is connected to a TTY device, otherwise returns false.
5	File.next()	It returns the next line from the file.
6	File.read([size])	It reads the file for the specified size.
7	File.readline([size])	It reads one line from the file and places the file pointer to the beginning of the new line.
8	File.readlines([sizehint])	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
9	File.seek(offset[,from])	It modifies the position of the file pointer to a specified offset with the specified reference.
10	File.tell()	It returns the current position of the file pointer within the file.
11	File.truncate([size])	It truncates the file to the optional specified size.
12	File.write(str)	It writes the specified string to a file
13	File.writelines(seq)	It writes a sequence of the strings to a file.

Using loops to process files Processing Records

Using a For Loop

The most common way to iterate over files in a directory using Python is by using a for loop. To use a for loop to iterate over files in a directory, we first need to use the `os.listdir()` function to get a [list](#) of all files in the directory. We can then use the `for` statement to loop over each file and perform the desired operation.

Example:

```
import os

directory = '/my_directory'
for filename in os.listdir(directory):
    if filename.endswith('.txt'):
        with open(os.path.join(directory, filename)) as f:
            print(f.read())
```

In the Python code above, we first define the directory containing the files that we want to iterate over. We then use the `os.listdir()` function to get a [list of all files in the directory](#). We then use a for loop to loop over each file in the directory. We use the if statement to check if the file has a “.txt” extension, and if it does, we open the file and print its content

Using a While Loop

Another way to iterate over files in Python is by using a while loop. To use a while loop to iterate over files in a directory, we first need to use the `os.listdir()` function to get a list of all files in the directory. We can then use a while loop to loop over each file and perform the desired operation.

Example:

```
import os

directory = '/my_directory'
files = os.listdir(directory)
index = 0
while index < len(files):
    filename = files[index]
    if filename.endswith('.txt'):
        with open(os.path.join(directory, filename)) as f:
            print(f.read())
    index += 1
```

In the code above, we first define the directory that we want to iterate over. We then use the `os.listdir()` function to get a list of all files in the directory. We then use a while loop to loop over each file in the directory. We use the if statement to check if the file has a “.txt” extension, and if it does, we open the file and print its contents. We also use an index variable to keep track of the current file being processed.

Using the os module

The `os` module in Python provides several functions for working with files and directories. One of these functions is [os.walk\(\)](#), which we can use to iterate over files in a directory. The `os.walk()` function traverses a directory tree and returns a tuple of the current directory, all subdirectories, and all filenames in the current directory.

Example:


```
import os

directory = '/my_directory'
for dirpath, dirnames, filenames in os.walk(directory):
    for filename in filenames:
        if filename.endswith('.txt'):
            with open(os.path.join(dirpath, filename)) as f:
                print(f.read())
```

In the code above, we first define the directory that we want to iterate over. We then use the `os.walk()` function to traverse the directory tree and get a tuple of the current directory, subdirectories, and filenames. We then use a nested for loop to loop over each filename and check if it has a “.txt” extension. If it does, we open the file and print its contents.

Using the glob module

The [glob module in Python](#) provides a function for working with file paths. The `glob.glob()` function allows us to search for files in a directory using a pattern. We can use the `glob.glob()` function to iterate over files in a directory by specifying a pattern that matches the files we want to process.

Example:

```
import glob

directory = '/my_directory'
for filename in glob.glob(directory + '/*.txt'):
    with open(filename) as f:
        print(f.read())
```

Using the pathlib module

The [pathlib module in Python](#) provides a path object that we can use to work with file paths.

The `pathlib.Path()` class provides several methods for working with files and directories, including `glob()`, which we can use to iterate over files in a directory.

Example:

```
import pathlib

directory = '/my_directory'
for path in pathlib.Path(directory).glob('*.txt'):
    with open(str(path)) as f:
        print(f.read())
```

Exception in Python

Exceptions are raised when some internal events occur which change the normal flow of the program.

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Example: Let us try to access the array element whose index is out of bound and handle the corresponding exception.

Python3

```
# Python program to handle simple runtime error
#Python 3
a = [1, 2, 3]
try:
    print ("Second element = %d" %(a[1]))
    # Throws error since there are only 3 elements in array
    print ("Fourth element = %d" %(a[3]))

except:
    print ("An error occurred")
```

Output

```
Second element = 2
An error occurred
```

NOTE

In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add `IndexError` in the above code. The general syntax for adding specific exceptions are –

try:

```
# statement(s)
```

except `IndexError`:

```
# statement(s)
```

except `ValueError`:

statement(s)

Example: Catching specific exceptions in the Python

```
# except statement
# Python 3

def fun(a):
    if a < 4:
        # throws ZeroDivisionError for a = 3
        b = a/(a-3)

    # throws NameError if a >= 4
    print("Value of b = ", b)

try:
    fun(3)
    fun(5)

# note that braces () are necessary here for
# multiple exceptions
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

Output

```
ZeroDivisionError Occurred and Handled
```

Try with Else Clause

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Example: Try with else clause

Python3

```
# Program to depict else clause with try-except
# Python 3
# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)

# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Output:

```
-5.0
a/b result in 0
```

Finally Keyword in Python

Python provides a keyword [finally](#), which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

Syntax:

try:

```
# Some Code....
```

except:

```
# optional block
```

```
# Handling of exception (if required)
```





else:

```
# execute if no exception
```

finally:

```
# Some code .....(always executed)
```

Python3

```
 # Python program to demonstrate finally  
 # No exception Exception raised in try block  
try:  
     k = 5//0 # raises divide by zero exception.  
     print(k)  
 # handles zerodivision exception  
except ZeroDivisionError:  
     print("Can't divide by zero")  
  
finally:  
     # this block is always executed  
     # regardless of exception generation.  
     print('This is always executed')
```

Output:

```
Can't divide by zero  
This is always executed
```

Raising Exception

The [raise statement](#) allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

Python3

```
# Program to depict Raising Exception
try:
    raise NameError("Hi there") # Raise Error
except NameError:
    print ("An exception")
    raise # To determine whether the exception was raised or not
```

The output of the above code will simply line printed as "An exception" but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

```
Traceback (most recent call last):
  File "/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line 5, in
<module>
    raise NameError("Hi there") # Raise Error
NameError: Hi there
```

Advantages of Exception Handling:

- **Improved program reliability:** By handling exceptions properly, you can prevent your program from crashing or producing incorrect results due to unexpected errors or input.
- **Simplified error handling:** Exception handling allows you to separate error handling code from the main program logic, making it easier to read and maintain your code.

- **Cleaner code:** With exception handling, you can avoid using complex conditional statements to check for errors, leading to cleaner and more readable code.
- **Easier debugging:** When an exception is raised, the Python interpreter prints a traceback that shows the exact location where the exception occurred, making it easier to debug your code.

Disadvantages of Exception Handling:

- **Performance overhead:** Exception handling can be slower than using conditional statements to check for errors, as the interpreter has to perform additional work to catch and handle the exception.
- **Increased code complexity:** Exception handling can make your code more complex, especially if you have to handle multiple types of exceptions or implement complex error handling logic.
- **Possible security risks:** Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it's important to handle exceptions carefully and avoid exposing too much information about your program.

Classes and Objects in Python:

Classes in Python:

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

Creating Classes in Python

In Python, a class can be created by using the keyword `class`, followed by the class name. The syntax to create a class is given below.

Syntax

```
class ClassName:  
    #statement_suite
```


Example:

```
class Person:
    def __init__(self, name, age):
        # This is the constructor method that is called when creating a new Person object
        # It takes two parameters, name and age, and initializes them as attributes of the object
        self.name = name
        self.age = age
    def greet(self):
        # This is a method of the Person class that prints a greeting message
        print("Hello, my name is " + self.name)
```

Objects in Python:

An object is a particular instance of a class with unique characteristics and functions. After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python. The object's attributes are initialised in the constructor, which is a special procedure with the name `__init__`.

Syntax:

```
# Declare an object of a class
object_name = Class_Name(arguments)
```

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        print("Hello, my name is " + self.name)

# Create a new instance of the Person class and assign it to the variable person1
person1 = Person("Ayan", 25)
person1.greet()
```

Output:

```
"Hello, my name is Ayan"
```

Python OOPs Concepts

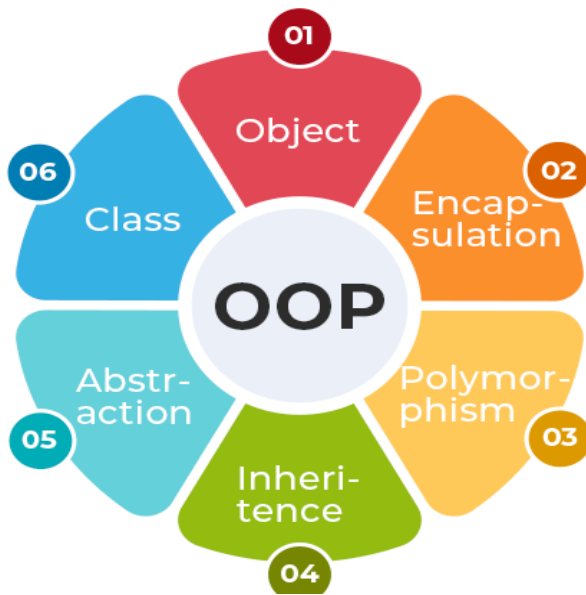
Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach. In **Python**, we can easily create and use classes and objects.

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-world entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects.

Major principles of object-oriented programming system are given below

- Class
- Object
- Method

- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation



Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax

class ClassName:

<statement-1>

.

.

<statement-N>

Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute `__doc__`, which returns the docstring defined in the function source code.

When we define a class, it needs to create an object to allocate the memory. Consider the following example.

Example:

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year
    def display(self):
        print(self.modelname,self.year)
c1 = car("Toyota", 2016)
c1.display()
```

Output:

```
Toyota 2016
```

In the above example, we have created the class named car, and it has two attributes modelname and year. We have created a c1 object to access the class attribute.

Method

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

Inheritance

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides the re-usability of the code.

Polymorphism

Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape. By polymorphism, we understand that one task can be performed in different ways. For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Encapsulation

Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

Protected Members

Protected members in C++ and Java are members of a class that can only be accessed within the class but cannot be accessed by anyone outside it. This can be done in Python by following the convention and prefixing the name with a single underscore.

The protected variable can be accessed from the class and in the derived classes (it can also be modified in the derived classes), but it is customary to not access it out of the class body.

The `__init__` method, which is a constructor, runs when an object of a type is instantiated.

Example:

```
# Python program for demonstrating protected members

# first, we will create the base class
class Base1:
    def __init__(self):
        # the protected member
        self._p = 78
# here, we will create the derived class
class Derived1(Base1):
    def __init__(self):
# now, we will call the constructor of Base class
        Base1.__init__(self)
        print ("We will call the protected member of base class: ",
              self._p)

# Now, we will be modifying the protected variable:
self._p = 433
        print ("we will call the modified protected member outside the class: ",
              self._p)
obj_1 = Derived1()
obj_2 = Base1()
# here, we will call the protected member
# this can be accessed but it should not be done because of convention
print ("Access the protected member of obj_1: ", obj_1._p)
# here, we will access the protected variable outside
print ("Access the protected member of obj_2: ", obj_2._p)
```

Output:

```
We will call the protected member of base class: 78
we will call the modified protected member outside the class: 433
Access the protected member of obj_1: 433
Access the protected member of obj_2: 78
```

Private Members

Private members are the same as protected members. The difference is that class members who have been declared private should not be accessed by anyone outside the class or any base classes. Python does not have Private instance variable variables that can be accessed outside of a class.

However, to define a private member, prefix the member's name with a double underscore "__".

Python's private and secured members can be accessed from outside the class using Python name mangling.

Example:

```
class Base1:
    def __init__(self):
        self.p = "Javatpoint"
        self.__q = "Javatpoint"
# Creating a derived class
class Derived1(Base1):
    def __init__(self):
# Calling constructor of
# Base class
        Base1.__init__(self)
        print("We will call the private member of base class: ")
        print(self.__q)
# Driver code
obj_1 = Base1()
print(obj_1.p)
```

Output:

```
Javatpoint
```

Polymorphism in Python

Polymorphism refers to having multiple forms. Polymorphism is a programming term that refers to the use of the same function name, but with different signatures, for multiple types.

Example

```
x = "Hello World!"
```

```
print(len(x))
```

```
mytuple = ("apple", "banana", "cherry")
```

```
print(len(mytuple))
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

```
print(len(thisdict))
```

Output

```
12
```

```
3
```

```
3
```

Classes in Python:

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

Creating Classes in Python

In Python, a class can be created by using the keyword `class`, followed by the class name. The syntax to create a class is given below.

Syntax

```
class ClassName:  
    #statement_suite
```

Note:

A class contains a statement suite including fields, constructor, function, etc. definition.

Example:

```
class Person:  
    def __init__(self, name, age):  
        # This is the constructor method that is called when creating a new Person object  
        # It takes two parameters, name and age, and initializes them as attributes of the object  
        self.name = name  
        self.age = age  
    def greet(self):  
        # This is a method of the Person class that prints a greeting message  
        print("Hello, my name is " + self.name)
```

Note:

Name and age are the two properties of the Person class. Additionally, it has a function called `greet` that prints a greeting.

Objects in Python:

An object is a particular instance of a class with unique characteristics and functions. After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python. The object's attributes are initialised in the constructor, which is a special procedure with the name `__init__`.

Syntax:

```
# Declare an object of a class
object_name = Class_Name(arguments)
```

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        print("Hello, my name is " + self.name)
# Create a new instance of the Person class and assign it to the variable person1
person1 = Person("Ayan", 25)
person1.greet()
```

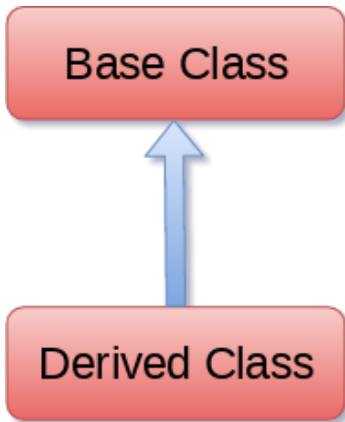
Output:

```
"Hello, my name is Ayan"
```

Python Inheritance

Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class



Syntax

```
class derived-class(base class):
```

```
<class-suite>
```

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Syntax

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
```

```
<class - suite>
```

Example 1

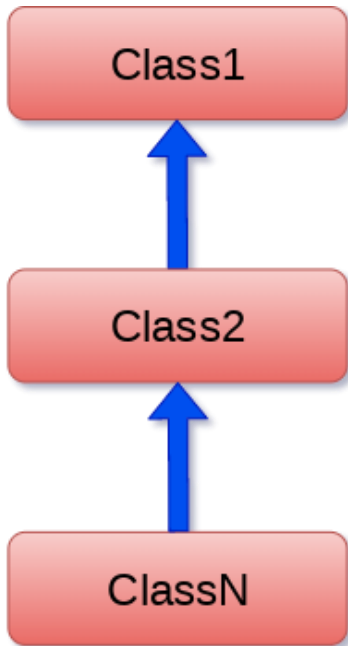
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is achieved when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is achieved in python.



Syntax

class class1:

<class-suite>

class class2(class1):

<class suite>

class class3(class2):

<class suite>

.

.

Example

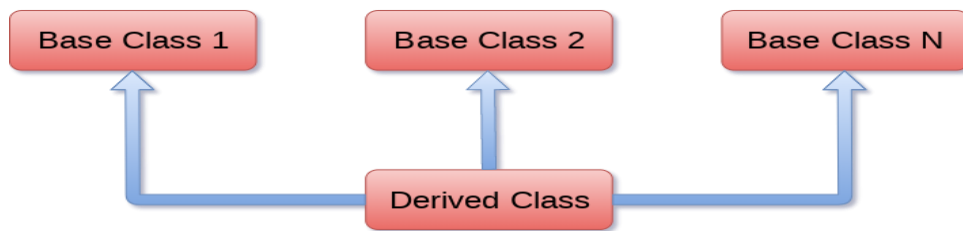
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



Syntax

```
class Base1:
```

```
    <class-suite>
```

```
class Base2:
```

```
    <class-suite>
```

```
•
```

```
•
```

```
•
```

```
class BaseN:
```

```
    <class-suite>
```

```
class Derived(Base1, Base2, ..... BaseN):
```

```
    <class-suite>
```

Example

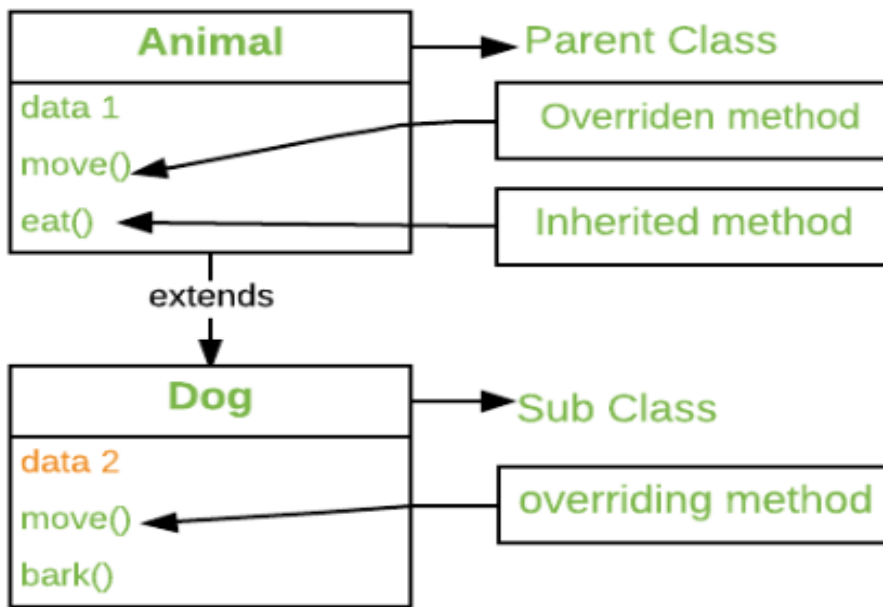
```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

Method Overriding in Python

Any object-oriented programming language can allow a subclass or child class to offer a customized implementation of a method already supplied by one of its superclasses or parent classes. This capability is known as method overriding. The term "override" refers to a method in a subclass that replaces a method in a superclass when both methods share the same name, parameters, signature, and return type (or sub-type).



The object that calls a method will determine which version of the method is executed. When a method is called from an object of a parent class, the method's parent class version is executed; however, when a method is called from an object of a subclass, the child class version is executed. In other words, the version of an overridden method depends on the object being referenced, not the type of the reference variable.

Example

```

# producer
def __init__(self):
    self.value = "Inside Parent"
# showing a parents method
def show(self):
    print(self.value)
# Defining a child class
class Child(Parent):
    # Constructor
    def __init__(self):
        self.value = "Inside Child"
# showing the child's method
def show(self):
    print(self.value)
# Chauffeur's code
obj1 = Parent()
obj2 = Child()
obj1.show()
obj2.show()

```

Output:

```

Inside Parent
Inside Child

```

Encapsulation in Python

Encapsulation is one of the most fundamental concepts in object-oriented programming (OOP). This is the concept of wrapping data and methods that work with data in one unit. This prevents data modification accidentally by limiting access to variables and methods. An object's method can change a variable's value to prevent accidental changes. These variables are called private variables.

Encapsulation is demonstrated by a class which encapsulates all data, such as member functions, variables, and so forth.

*Protected Members #refer Data Abstraction

*Private Members

Data Hiding in Python

Data hiding is a part of object-oriented programming, which is generally used to hide the data information from the user. It includes internal object details such as data members, internal working. It maintained the data integrity and restricted access to the class member. The main working of data hiding is that it combines the data and functions into a single unit to conceal data within a class. We cannot directly access the data from outside the class.

Example -

```
class CounterClass:
    __privateCount = 0
    def count(self):
        self.__privateCount += 1
        print(self.__privateCount)
counter = CounterClass()
counter.count()
counter.count()
print(counter.__privateCount)
```

Output:

```
1
2
Traceback (most recent call last):
  File "<string>", line 17, in <module>
AttributeError: 'CounterClass' object has no attribute '__privateCount'
```

UNIT IV

DATA MANIPULATION TOOLS & SOFTWARES

NumPy

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

Use NumPy

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

NumPy: Installation

To run the NumPy program, first, Numpy needs to be installed. Numpy is installed from [Python's official website](#) using pip and conda by running different commands on different operating systems. Many important packages are automatically installed within the Numpy library.

NOTE:

It is mandatory to install **Python** from [Python's official website](#) before installing NumPy.

Installing Numpy on Windows:

Steps for installing NumPy on Windows:

1. **Install NumPy** using the following PIP command in the command prompt terminal:

```
pip install numpy
```



```
Command Prompt
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\durgu> pip install numpy
```

The installation will start automatically, and Numpy will be successfully installed with its latest version.



```
Command Prompt
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

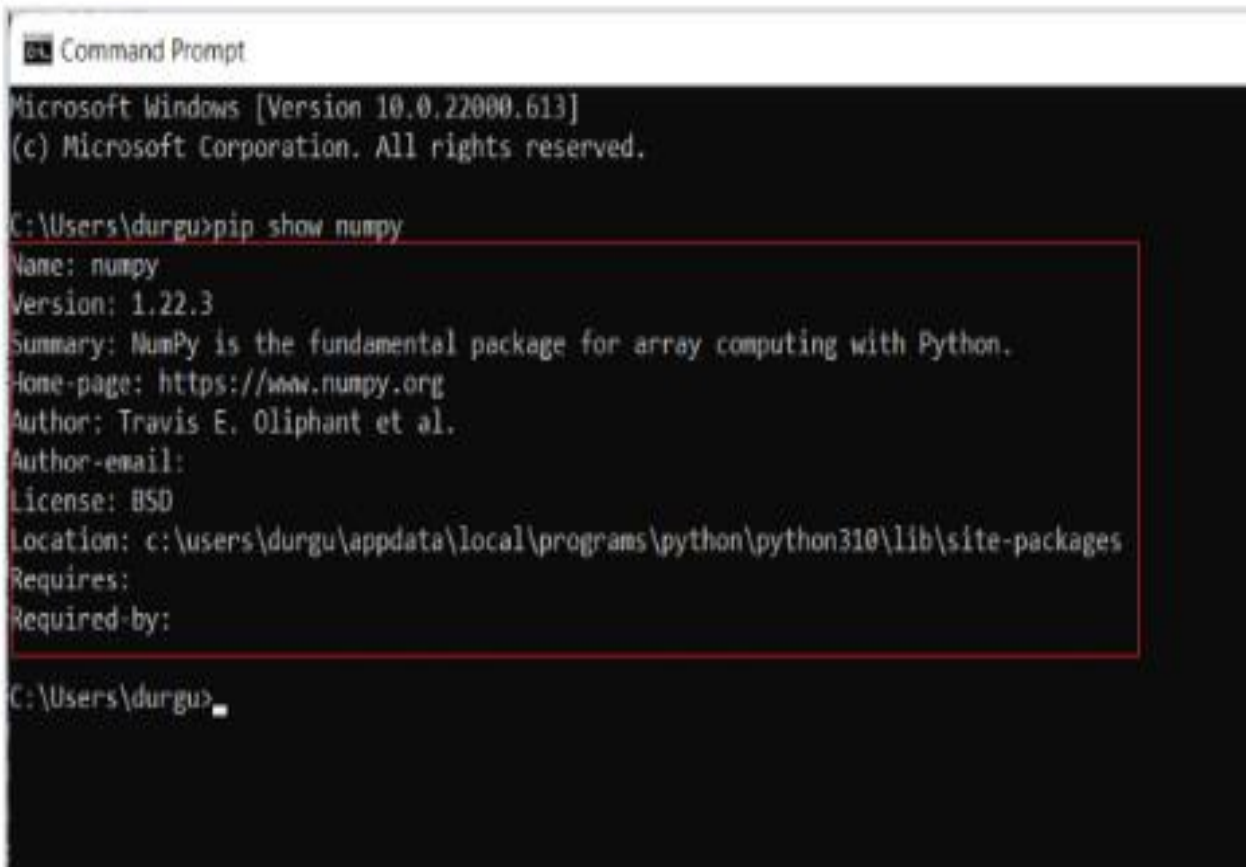
C:\Users\durgu> pip install numpy
Collecting numpy
  Downloading numpy-1.22.3-cp310-cp310-win_amd64.whl (14.7 MB)
----- 14.7/14.7 MB 4.5 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-1.22.3

C:\Users\durgu>
```

2. **Verify NumPy Installation** by typing the command given below in cmd:

```
pip show Numpy
```

It will show the location and numpy version installed.



```
Command Prompt
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\durgu>pip show numpy
Name: numpy
Version: 1.22.3
Summary: NumPy is the fundamental package for array computing with Python.
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email:
License: BSD
Location: c:\users\durgu\appdata\local\programs\python\python310\lib\site-packages
Requires:
Required-by:

C:\Users\durgu>
```

3. Import NumPy Package

1. Create python Environment in cmd by typing:**Python**

```
Command Prompt - python
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\durgu>python
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
>>>
```

2. Type command **import numpy as np**

4. Upgrade NumPy by using the following command:

```
pip install --upgrade numpy
```

```
Command Prompt
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\durgu>pip install --upgrade numpy
```

Ndarray

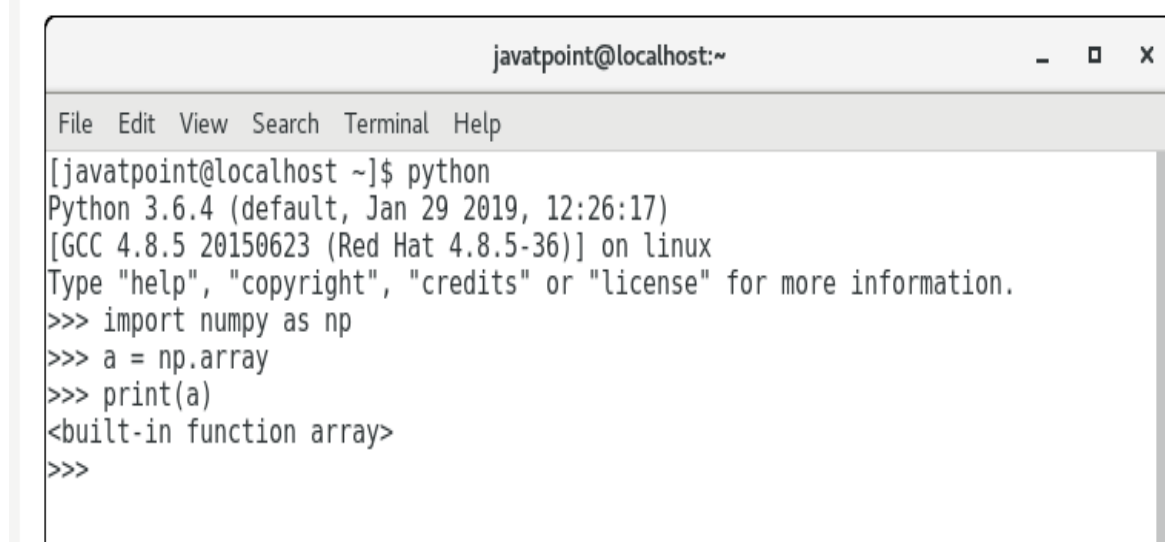
Ndarray is the n-dimensional array object defined in the numpy which stores the collection of the similar type of elements. In other words, we can define ndarray as the collection of the data type (dtype) objects.

The ndarray object can be accessed by using the 0 based indexing. Each element of the Array object contains the same size in the memory.

Creating a ndarray object

The ndarray object can be created by using the array routine of the numpy module. For this purpose, we need to import the numpy.

```
>>> a = numpy.array
```

A screenshot of a terminal window titled 'jvatpoint@localhost:~'. The terminal shows the following text:

```
[jvatpoint@localhost ~]$ python
Python 3.6.4 (default, Jan 29 2019, 12:26:17)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a = np.array
>>> print(a)
<built-in function array>
>>>
```

We can also pass a collection object into the array routine to create the equivalent n-dimensional array. The syntax is given below.

```
>>> numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)
```

The parameters are described in the following table.

SN	Parameter	Description
1	object	It represents the collection object. It can be a list, tuple, dictionary, set, etc.
2	dtype	We can change the data type of the array elements by changing this option to the specified type. The default is none.
3	copy	It is optional. By default, it is true which means the object is copied.
4	order	There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any)
5	subok	The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true.
6	ndmin	It represents the minimum dimensions of the resultant array.

To create an array using the list, use the following syntax.

```
>>> a = numpy.array([1, 2, 3])
```

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python  
Python 3.6.4 (default, Jan 29 2019, 12:26:17)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import numpy  
>>> a = numpy.array([[1,2,3],[4,5,6]])  
>>> print(a)  
[[1 2 3]  
 [4 5 6]]  
>>> █
```

To create a multi-dimensional array object, use the following syntax.

```
>>> a = numpy.array([[1, 2, 3], [4, 5, 6]])
```

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python  
Python 3.6.4 (default, Jan 29 2019, 12:26:17)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import numpy  
>>> a = numpy.array([1,2,3])  
>>> print(a)  
[1 2 3]  
>>> print(type(a))  
<class 'numpy.ndarray'>  
>>> █
```

To change the data type of the array elements, mention the name of the data type along with the collection.

```
>>> a = numpy.array([1, 3, 5, 7], complex)
```

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python  
Python 3.6.4 (default, Jan 29 2019, 12:26:17)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import numpy  
>>> a = numpy.array([1, 3, 5, 7],dtype = complex)  
>>> print(a)  
[1.+0.j 3.+0.j 5.+0.j 7.+0.j]  
>>> █
```

Finding the dimensions of the Array

The **ndim** function can be used to find the dimensions of the array.

```
>>> import numpy as np  
>>> arr = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [9, 10, 11, 23]])  
>>> print(arr.ndim)
```

```
javatpoint@localhost:~  
File Edit View Search Terminal Help  
[javatpoint@localhost ~]$ python  
Python 3.6.4 (default, Jan 29 2019, 12:26:17)  
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import numpy as np  
>>> arr = np.array([[1,2,3,4],[4,5,6,7],[9,10,11,23]])  
>>> print(arr.ndim)  
2  
>>> print(arr)  
[[ 1  2  3  4]  
 [ 4  5  6  7]  
 [ 9 10 11 23]]  
>>> █
```

NumPy Array Slicing

Array Slicing is the process of extracting a portion of an array.

With slicing, we can easily access elements in the array. It can be done on one or more dimensions of a NumPy array.

Syntax

```
array[start:stop:step]
```

Here,

- start - index of the first element to be included in the slice
- stop - index of the last element (exclusive)
- step - step size between each element in the slice

Note: When we slice arrays, the start index is inclusive but the stop index is exclusive.

- If we omit start, slicing starts from the first element
- If we omit stop, slicing continues up to the last element
- If we omit step, default step size is 1

```
import numpy as np

# create a 1D array
array1 = np.array([1, 3, 5, 7, 8, 9, 2, 4, 6])

# slice array1 from index 2 to index 6 (exclusive)
print(array1[2:6]) # [5 7 8 9]

# slice array1 from index 0 to index 8 (exclusive) with a step size of 2
print(array1[0:8:2]) # [1 5 8 2]

# slice array1 from index 3 up to the last element
print(array1[3:]) # [7 8 9 2 4 6]

# items from start to end
print(array1[:]) # [1 3 5 7 8 9 2 4 6]
```

Output

```
[5 7 8 9]
```

```
[1 5 8 2]
```

```
[7 8 9 2 4 6]
```

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

```
>
```

Array indexing

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4])
```

```
print(arr[0])
```

Output

```
>1
```

Example

Get the second element from the following array.

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4])
```

```
print(arr[1])
```

Output

>2

NumPy Array Iterating

Iterating means going through elements one by one.

Example

Iterate on the elements of the following 1-D array:

```
import numpy as np
```

```
arr = np.array([1, 2, 3])
```

```
for x in arr:
```

```
    print(x)
```

Output

>1

2

3

Shape Manipulation

The `numpy.reshape()` function is used to change the shape (dimensions) of an array without changing its data. This function returns a new array with the same data but with a different shape.

The `numpy.reshape()` function is useful when we need to change the dimensions of an array, for example, when we want to convert a one-dimensional array into a two-dimensional array or vice versa. It can also be used to create arrays with a specific shape, such as matrices and tensors.

Syntax:

```
numpy.reshape(a, newshape, order='C')
```

Here

A - Array to be reshaped

newshape - The new shape should be compatible with the original shape

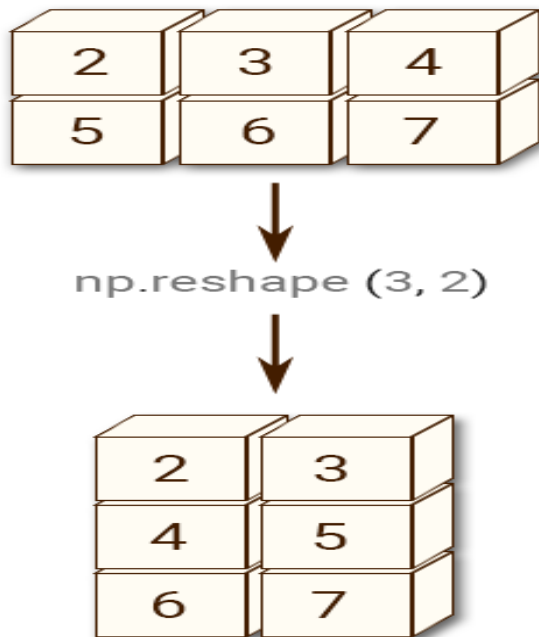
order - Read the elements of a using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using

C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order.

Example

```
>>> import numpy as np
>>> x = np.array([[2,3,4], [5,6,7]])
>> np.reshape(x, (3, 2))
array([[2, 3],
       [4, 5],
       [6, 7]])
```

Output



Array Manipulation

Arrays are used to store multiple values in one single variable.

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

```
print(cars)
```

Output

```
["Ford", "Volvo", "BMW"]
```

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
x = cars[0]
```

Output

```
Ford
```

Example

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

```
print(cars)
```

Output

```
["Toyota", "Volvo", "BMW"]
```

The Length of an Array

Use the `len()` method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the `cars` array:

```
x = len(cars)
```

```
print(x)
```

Output

3

Looping Array Elements

You can use the `for in` loop to loop through all the elements of an array.

Example

Print each item in the `cars` array:

```
for x in cars:
```

```
    print(x)
```

Output

Toyota

Volvo

BMW

Adding Array Elements

You can use the `append()` method to add an element to an array.

Example

Add one more element to the `cars` array:

```
cars.append("Honda")
```

```
print(cars)
```

Output

```
["Ford", "Volvo", "BMW", "Honda"]
```

Removing Array Elements

You can use the `pop()` method to remove an element from the array.

Example

Delete the second element of the `cars` array:

```
cars.pop(1)
```

```
print(cars)
```

Output

```
["Ford", "BMW", "Honda"]
```

You can also use the `remove()` method to remove an element from the array.

Example

Delete the element that has the value "Volvo":

```
cars.remove("Volvo")
```

```
print(cars)
```

Output

```
["Ford", "BMW", "Honda"]
```

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the first item with the specified value
<u>reverse()</u>	Reverses the order of the list

[sort\(\)](#)

Sorts the list

Ndarray Manipulation

Changing Shape

Sr.No.	Shape & Description
1	reshape Gives a new shape to an array without changing its data
2	flat A 1-D iterator over the array
3	flatten Returns a copy of the array collapsed into one dimension
4	ravel Returns a contiguous flattened array

Transpose Operations

Sr.No.	Operation & Description
1	transpose Permutes the dimensions of an array
2	ndarray.T Same as self.transpose()
3	rollaxis Rolls the specified axis backwards
4	swapaxes Interchanges the two axes of an array

Changing Dimensions

Sr.No.	Dimension & Description
1	broadcast Produces an object that mimics broadcasting
2	broadcast_to Broadcasts an array to a new shape
3	expand_dims Expands the shape of an array
4	squeeze Removes single-dimensional entries from the shape of an array

Joining Arrays

Sr.No.	Array & Description
1	concatenate Joins a sequence of arrays along an existing axis
2	stack Joins a sequence of arrays along a new axis
3	hstack Stacks arrays in sequence horizontally (column wise)
4	vstack Stacks arrays in sequence vertically (row wise)

Splitting Arrays

Sr.No.	Array & Description
1	split Splits an array into multiple sub-arrays
2	hsplit Splits an array into multiple sub-arrays horizontally (column-wise)
3	vsplit Splits an array into multiple sub-arrays vertically (row-wise)

Adding / Removing Elements

Sr.No.	Element & Description
1	resize Returns a new array with the specified shape
2	append Appends the values to the end of an array
3	insert Inserts the values along the given axis before the given indices
4	delete Returns a new array with sub-arrays along an axis deleted
5	unique Finds the unique elements of an array

Numpy's Structured Array

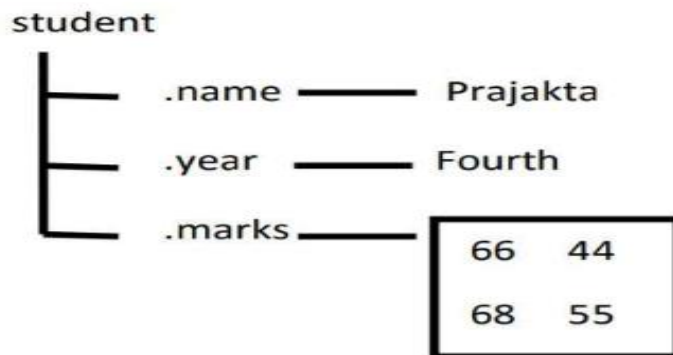
Numpy's Structured Array is similar to Struct in C. It is used for grouping data of different types and sizes. Structure array uses data containers called fields. Each data field can contain data of any type and size. Array elements can be accessed with the help of dot notation.

Note: Arrays with named fields that can contain data of various types and sizes.

Properties of Structured array

- All structs in array have same number of fields.
- All structs have same fields names.

For example, consider a structured array of student which has different fields like name, year, marks.



Each record in array student has a structure of class Struct. The array of a structure is referred to as struct as adding any new fields for a new struct in the array, contains the empty array.

Example

```
# Python program to demonstrate
```

```
# Structured array
```

```
import numpy as np
```

```
a = np.array([('Sana', 2, 21.0), ('Mansi', 7, 29.0)],
```

```
dtype=[('name', (np.str_, 10)), ('age', np.int32), ('weight', np.float64)])
```

```
print(a)
```

Output:

```
[('Sana', 2, 21.0) ('Mansi', 7, 29.0)]
```

Reading and writing array data on files in python using NumPy

Reading array data on files in python using NumPy

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object and tools for working with these arrays.

Numerical data can be present in different formats of file :

- The data can be saved in a txt file where each line has a new data point.
- The data can be stored in a CSV(comma separated values) file.
- The data can be also stored in TSV(tab separated values) file.

```
Syntax: numpy.loadtxt(fname, dtype='float', comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0)
```

Example 1: Reading numerical data from text file

```
# Importing libraries that will be used
```

```
import numpy as np
```

```
# Setting name of the file that the data is to be extracted from in python
```

```
filename = 'gfg_example1.txt'
```

```
# Loading file data into numpy array and storing it in variable called data_collected
```

```
data_collected = np.loadtxt(filename)
```

```
# Printing data stored
```

```
print(data_collected)
```

```
# Type of data
```



```
print( f'Stored in : {type(data_collected)} and data type is : {data_collected.dtype}')
```

Output

```
[1.0000e+00 2.0000e+00 3.0000e+00 4.0000e+00 5.0000e+00 6.0000e+00  
7.0000e+00 8.0000e+00 9.0000e+00 1.0000e+02 1.2100e+02 1.2345e+04  
1.2000e+01 1.6320e+03 9.0100e+02]  
Stored in : <class 'numpy.ndarray'> and the data type is : float64
```

Writing array data on files in python using NumPy

The Numpy array can be saved to a text file using various methods like the `savetxt()` method, `save()` method, and `dataframe.to_csv()` function. Numpy is a Python library that is used to do the numerical computation, manipulate arrays, etc.

Method 1: Using the `numpy.savetxt()` function

The `numpy.savetxt()` function simply saves the numpy array to a text file. The function takes two arguments - the file name in which the numpy array is to be saved and the array itself.

In the below example, we create a 2D NumPy array `arr` and save it to a text file called `'myarray.txt'` using the `numpy.savetxt()` function. The delimiter character used in the text file is a space, which is the default delimiter. You can change the delimiter by specifying the `'delimiter'` parameter in the `numpy.savetxt()` function.

```
import numpy as np  
  
# Create a 2D NumPy array  
arr = np.array([[1, 2, 3], [4, 5, 6]])  
  
# Save the array to a text file  
np.savetxt('myarray.txt', arr)
```

Output

```
1.0000000000000000e+00 2.0000000000000000e+00 3.0000000000  
4.0000000000000000e+00 5.0000000000000000e+00 6.0000000000
```

Method 2: Using the `numpy.save()` Function

The `numpy.save()` function saves the array to a binary file with the ``.numpy`` extension. The file can be loaded back to Python using the `numpy.load()` function.

Example

In the below example, we create a NumPy array `arr` and save it to a binary file called `'myarray.npy'` using the `numpy.save()` function. The resulting file is a binary file that can be loaded back into Python using the `numpy.load()` function.

```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Save the array to a binary file
np.save('myarray.npy', arr)

# Load the saved array
arr_loaded = np.load('myarray.npy')
print(arr_loaded)
```

Edit & Run 

Output

```
[1 2 3 4 5]
```

Pandas

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

Key Features of Pandas

- It has a DataFrame object that is quick and effective, with both standard and custom indexing.
- Utilized for reshaping and turning of the informational indexes.
- For aggregations and transformations, group by data.
- It is used to align the data and integrate the data that is missing.
- Provide Time Series functionality.
- Process a variety of data sets in various formats, such as matrix data, heterogeneous tabular data, and time series.

- Manage the data sets' multiple operations, including subsetting, slicing, filtering, groupBy, reordering, and reshaping.
- It incorporates with different libraries like SciPy, and scikit-learn.
- Performs quickly, and the Cython can be used to accelerate it even further.

Benefits of Pandas

The following are the advantages of pandas overusing other languages:

Representation of Data: Through its DataFrame and Series, it presents the data in a manner that is appropriate for data analysis.

Clear code: Pandas' clear API lets you concentrate on the most important part of the code. In this way, it gives clear and brief code to the client.

Installation of Pandas

If you have [Python](#) and [PIP](#) already installed on a system, then installation of Pandas is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install pandas
```

If this command fails, then use a python distribution that already has Pandas installed like, Anaconda, Spyder etc.

Import Pandas

Once Pandas is installed, import it in your applications by adding the `import` keyword:

```
import pandas
```

Now Pandas is imported and ready to use.

Example [Get your own Python Server](#)

```
import pandas
```

```
mydataset = {  
    'cars': ["BMW", "Volvo", "Ford"],  
    'passings': [3, 7, 2]  
}
```

```
myvar = pandas.DataFrame(mydataset)
```

```
print(myvar)
```

Output

```
cars  passings  
0  BMW        3  
1  Volvo      7  
2  Ford       2
```

Pandas as pd

Pandas is usually imported under the `pd` alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the `as` keyword while importing:

```
import pandas as pd
```

Now the Pandas package can be referred to as `pd` instead of `pandas`.

Example

```
import pandas as pd
```

```
mydataset = {  
    'cars': ["BMW", "Volvo", "Ford"],  
    'passings': [3, 7, 2]  
}
```

```
myvar = pd.DataFrame(mydataset)
```

```
print(myvar)
```

Output

```
cars  passings
0  BMW      3
1  Volvo    7
2  Ford     2
```

Introduction to pandas Data Structures

Pandas deals with the following three data structures –

- Series
- DataFrame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Building and handling two or more dimensional arrays is a tedious task, burden is placed on the user to consider the orientation of the data set when writing functions. But using Pandas data structures, the mental effort of the user is reduced.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1.

Mutability

All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.

Note – DataFrame is widely used and one of the most important data structures. Panel is used much less.

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, ...

10 23 56 17 52 61 73 90 26 72

DataFrame

DataFrame is a two-dimensional array with heterogeneous data. For example,

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

Data Type of Columns

The data types of the four columns are as follows –

Column	Type
Name	String
Age	Integer
Gender	String
Rating	Float

Panel

Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

Operations between Data Structures in Pandas

In Pandas, there are different useful data operations for DataFrame, which are as follows

Row and column selection

We can select any row and column of the DataFrame by passing the name of the rows and column. When you select it from the DataFrame, it becomes one-dimensional and considered as Series.

Filter Data

We can filter the data by providing some of the boolean expression in DataFrame.

Null values

A Null value can occur when no data is being provided to the items. The various columns may contain no values which are usually represented as NaN. In Pandas, several useful functions are available for detecting, removing, and replacing the null values in Data Frame. These functions are as follows:

isnull(): The main task of isnull() is to return the true value if any row has null values.

notnull(): It is opposite of isnull() function and it returns true values for not null value.

dropna(): This method analyzes and drops the rows/columns of null values.

fillna(): It allows the user to replace the NaN values with some other values.

replace(): It is a very rich function that replaces a string, regex, series, dictionary, etc.

interpolate(): It is a very powerful function that fills null values in the DataFrame or series.

String operation

A set of a string function is available in Pandas to operate on string data and ignore the missing/NaN values. There are different string operation that can be performed using **.str.** option. These functions are as follows:

lower(): It converts any strings of the series or index into lowercase letters.

upper(): It converts any string of the series or index into uppercase letters.

strip(): This function helps to strip the whitespaces including a new line from each string in the Series/index.

split(' '): It is a function that splits the string with the given pattern.

cat(sep=' '): It concatenates series/index elements with a given separator.

contains(pattern): It returns True if a substring is present in the element, else False.

replace(a,b): It replaces the value a with the value b.

repeat(value): It repeats each element with a specified number of times.

count(pattern): It returns the count of the appearance of a pattern in each element.

startswith(pattern): It returns True if all the elements in the series starts with a pattern.

endswith(pattern): It returns True if all the elements in the series ends with a pattern.

find(pattern): It is used to return the first occurrence of the pattern.

findall(pattern): It returns a list of all the occurrence of the pattern.

swapcase: It is used to swap the case lower/upper.

islower(): It returns True if all the characters in the string of the Series/Index are in lowercase. Otherwise, it returns False.

isupper(): It returns True if all the characters in the string of the Series/Index are in uppercase. Otherwise, it returns False.

isnumeric(): It returns True if all the characters in the string of the Series/Index are numeric. Otherwise, it returns False.

Count Values

This operation is used to count the total number of occurrences using '**value_counts()**' option.

Plots

Pandas plots the graph with the **matplotlib** library. The **.plot()** method allows you to plot the graph of your data.

.plot() function plots index against every column.

You can also pass the arguments into the **plot()** function to draw a specific column.

Function Application and Mapping in Pandas

Function Application

The appropriate method for applying the functions depends on whether your function expects to operate element-wise, row wise, or column wise.

- **pipe():** Table wise function applications in Pandas
- **apply():** Row or column wise function operation

- **applymap()**: Element-wise function applications in Pandas

Before we explore the pandas function applications, we need to import pandas and numpy-

```
>>> import pandas as pd
```

```
>>> import numpy as np
```

Table-wise Function Application

Custom operations can be performed by passing the function and the appropriate number of parameters as pipe arguments. Thus, operation is performed on the whole DataFrame.

For example, add a value 2 to all the elements in the DataFrame. Then,

adder function

The adder function adds two numeric values as parameters and returns the sum.

```
def adder(ele1,ele2):  
    return ele1+ele2
```

We will now use the custom function to conduct operation on the DataFrame.

```
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])  
df.pipe(adder,2)
```

Let's see the full program –

[Live Demo](#)

```
import pandas as pd  
import numpy as np  
  
def adder(ele1,ele2):  
    return ele1+ele2
```

```
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.pipe(adder,2)
print df.apply(np.mean)
```

Its **output** is as follows –

```
   col1   col2   col3
0  2.176704  2.219691  1.509360
1  2.222378  2.422167  3.953921
2  2.241096  1.135424  2.696432
3  2.355763  0.376672  1.182570
4  2.308743  2.714767  2.130288
```

Row or Column Wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame or Panel using the **apply()** method, which, like the descriptive statistics methods, takes an optional axis argument. By default, the operation performs column wise, taking each column as an array-like.

Example 1

[Live Demo](#)

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.apply(np.mean)
print df.apply(np.mean)
```

Its **output** is as follows –

```
col1  -0.288022
col2   1.044839
col3  -0.187009
dtype: float64
```

By passing **axis** parameter, operations can be performed row wise.

Example 2

[Live Demo](#)

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.apply(np.mean,axis=1)
print df.apply(np.mean)
```

Its **output** is as follows –

```
col1  0.034093
col2 -0.152672
col3 -0.229728
dtype: float64
```

Example 3

[Live Demo](#)

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.apply(lambda x: x.max() - x.min())
print df.apply(np.mean)
```

Its **output** is as follows –

```
col1 -0.167413
col2 -0.370495
col3 -0.707631
dtype: float64
```

Element Wise Function Application

Not all functions can be vectorized (neither the NumPy arrays which return another array nor any value), the methods **applymap()** on DataFrame and **analogously map()** on Series accept any Python function taking a single value and returning a single value.

Example 1

[Live Demo](#)

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])

# My custom function
df['col1'].map(lambda x:x*100)
print df.apply(np.mean)
```

Its **output** is as follows –

```
col1  0.480742
col2  0.454185
col3  0.266563
dtype: float64
```

Example 2

[Live Demo](#)

```
import pandas as pd
import numpy as np

# My custom function
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df.applymap(lambda x:x*100)
print df.apply(np.mean)
```

Its **output** is as follows –

```
col1  0.395263
col2  0.204418
```

col3 -0.795188
dtype: float64

Sorting and Ranking in pandas

[pandas](#) is a powerful [Python](#) library for data manipulation and analysis, provides various functionalities to sort and rank data efficiently. It can be used for sorting and ranking organized data, identifying patterns, and making informed decisions.

Sorting

Sorting is rearranging data in ascending or descending order based on specific columns or rows. It is crucial for tasks like identifying the highest or lowest values, finding outliers, or preparing data for visualization.

Sorting can be done in multiple ways:

Sorting by columns

To sort a pandas DataFrame by a specific column, we can use the `sort_values()` method.

Syntax

```
sorted_df = df.sort_values(by='column_name', ascending=flag)
```

The parameters involved are as follows:

- `by`: Specifies the column by which the DataFrame should be sorted.
- `ascending`: Determines the sorting order. Set to `True` for ascending order and `False` for descending order. This parameter is optional, and if not specified, it defaults to `True`.

Code example

```
1 import pandas as pd
2
3 data = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie', 'David'],
4                      'Age': [25, 50, 35, 48],
5                      'Salary': [10000, 96000, 54000, 52000]})
6
7 sorted_data = data.sort_values(by='Age', ascending=True)
8 print(sorted_data)
```

Output

	Name	Age	Salary
0	Alice	25	10000
2	Charlie	35	54000
3	David	48	52000
1	Bob	50	96000

Sorting by rows

To sort the rows of a DataFrame based on their index or row labels, we can use the `sort_index()` method.

Syntax

```
sorted_df = df.sort_index(axis=0, ascending=flag)
```

The parameters involved are as follows:

- `axis`: Specifies the axis along which to sort. Set `axis=0` for rows and `axis=1` for columns.

- `ascending`: Determines the sorting order. Set to `True` for ascending order and `False` for descending order. This parameter is optional, and if not specified, it defaults to `True`.

Code example

```
1 import pandas as pd
2
3 data = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie', 'David'],
4                      'Age': [25, 50, 35, 48],
5                      'Salary': [10000, 96000, 54000, 52000]})
6
7 sorted_data = data.sort_index(axis=0, ascending=True)
8 print(sorted_data)
```

Output

	Name	Age	Salary
0	Alice	25	10000
1	Bob	50	96000
2	Charlie	35	54000
3	David	48	52000

Sorting by multiple columns

Sorting by multiple columns creates a hierarchical sorting order.

Syntax

```
sorted_df = df.sort_values(by=['column1', 'column2'], ascending=[flag_one, flag_two])
```

The parameters involved are as follows:

- `by`: Specifies a list of column names by which the DataFrame should be sorted. The sorting applies in the order the columns are listed.
- `ascending`: Determines the sorting order for each column. Set to `True` for ascending order and `False` for descending order. This parameter is optional, and if not specified, it defaults to `True` for all columns.

Code example

It sorts the DataFrame by `Name` in ascending order and then, within each `Name` group, by `Salary` in descending order.

```
1 import pandas as pd
2
3 data = pd.DataFrame({'Name': ['Alice', 'Bob', 'Alice', 'David'],
4                      'Age': [25, 50, 35, 48],
5                      'Salary': [10000, 96000, 54000, 52000]})
6
7 sorted_data = data.sort_values(by=['Name', 'Salary'], ascending=[True, False])
8 print(sorted_data)
```

Output

	Name	Age	Salary
2	Alice	35	54000
0	Alice	25	10000
1	Bob	50	96000
3	David	48	52000

Ranking

Ranking is assigning ranks or positions to data elements based on their values. This is particularly valuable when analyzing data with repetitive values or when you need to identify the top or bottom entries.

Syntax

```
df['Rank'] = df['column'].rank(axis=0, method='average')
```

The parameters involved are as follows:

- `axis`: Axis to rank. `0` for index and `1` for columns.

- **method**: Specifies the method used to rank data when there are ties (i.e., duplicate values). The available options are as follows:
 - **average (default)**: Assigns the average rank to tied values. For example, if two values have the same rank, they both get the average of the ranks they would have received if there were no ties.
 - **min**: Assigns the minimum rank to tied values. In the case of ties, the method assigns the smallest rank to all tied values.
 - **max**: Assigns the maximum rank to tied values. In the case of ties, the method assigns the largest rank to all tied values.
 - **first**: Assigns ranks in the order they appear in the data. The first occurrence of a value gets a rank of 1, the second occurrence gets a rank of 2, and so on.
 - **dense**: Similar to **'min'** but ranks are continuous without gaps. For example, if there are two tied values with ranks 2 and 3, both will receive a rank of 2.

Code example

We can customize the ranking behavior in the code by replacing the **'average'** parameter with one of the following options: **'min'**, **'max'**, **'first'**, or **'dense'** to observe different ranking outcomes.

```
1 import pandas as pd
2
3 data = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie', 'David'],
4                      'Age': [25, 35, 35, 48],
5                      'Salary': [10000, 96000, 54000, 52000]})
6
7 data['Rank'] = data['Age'].rank(method='average')
8 print(data)
```

Output

	Name	Age	Salary	Rank
0	Alice	25	10000	1.0
1	Bob	35	96000	2.5
2	Charlie	35	54000	2.5
3	David	48	52000	4.0

Correlation and Covariance in Pandas

Both covariance and correlation are about the relationship between the variables.

Covariance

[Covariance](#) measures how two variables change in relation to each other. In other words, it measures whether the variables increase or decrease together. If the variables tend to increase or decrease together, the covariance is positive. If one variable increases as the other decreases, then the covariance is negative.

$$cov_{x,y} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{N - 1}$$

The following represents the Python code for calculating the pairwise covariance of different columns of IRIS dataset. Note that IRIS dataset is loaded and a Pandas dataframe is created. Then, method `cov()` is invoked on the Pandas dataframe to calculate covariance.

```

1 import pandas as pd
2 from sklearn import datasets
3 #
4 # Load IRIS dataset
5 #
6 iris = datasets.load_iris()
7 #
8 # Create dataframe from IRIS dataset
9 #
10 df = pd.DataFrame(iris.data, columns=["sepal_length", "sepal_width",
11 "petal_length", "petal_width"])
12 df["class"] = iris.target
13 #
14 # Calculate covariance between different columns
15 #
    df.iloc[:, 0:4].cov()

```

	sepal_length	sepal_width	petal_length	petal_width
sepal_length	0.685694	-0.042434	1.274315	0.516271
sepal_width	-0.042434	0.189979	-0.329656	-0.121639
petal_length	1.274315	-0.329656	3.116278	1.295609
petal_width	0.516271	-0.121639	1.295609	0.581006

Correlation

[Correlation](#) is similar to covariance in that it measures how two variables change in relation to each other. However, correlation normalizes the variance of both variables, which makes it easier to interpret than covariance. Correlation can range from -1 to 1; a value of 0 indicates that there is no linear relationship between the two variables, a value of 1 indicates that there is a perfect positive linear relationship (i.e., as one variable increases, so does the other), and a value of -1 indicates that there is a perfect negative linear relationship (i.e., as one variable increases, the other decreases).

The formula for calculating correlation as a function of covariance and standard deviation goes as following:

$$\text{Correlation} = \frac{\text{Cov}(x, y)}{\sigma_x * \sigma_y}$$

The following represents the Python code for calculating the pairwise correlation of different columns of IRIS dataset. Note that IRIS dataset is loaded and a Pandas dataframe is created. Then, method `corr()` is invoked on the Pandas dataframe to calculate covariance.

```

1  import pandas as pd
2  from sklearn import datasets
3  #
4  # Load IRIS dataset
5  #
6  iris = datasets.load_iris()
7  #
8  # Create dataframe from IRIS dataset
9  #
10 df = pd.DataFrame(iris.data, columns=["sepal_length", "sepal_width",
11 "petal_length", "petal_width"])
12 df["class"] = iris.target
13 #
14 # Calculate pairwise correlation between different columns
15 #
    df.iloc[:, 0:4].corr()

```

	sepal_length	sepal_width	petal_length	petal_width
sepal_length	1.000000	-0.117570	0.871754	0.817941
sepal_width	-0.117570	1.000000	-0.428440	-0.366126
petal_length	0.871754	-0.428440	1.000000	0.962865
petal_width	0.817941	-0.366126	0.962865	1.000000

Not a Number Data

Missing Data can occur when no information is provided for one or more items or for a whole unit. Missing Data is a very big problem in a real-life scenarios. Missing Data can also refer to as NA(Not Available) values in pandas. In DataFrame sometimes many datasets simply arrive with missing data, either because it exists and was not collected or it never existed. For Example, Suppose different users being surveyed may choose not to share their income, some users may choose not to share the address in this way many datasets went missing.

In Pandas missing data is represented by two value:

- None: None is a Python singleton object that is often used for missing data in Python code.
- NaN : NaN (an acronym for Not a Number), is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation

there are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame

1. isnull
2. isnan
3. isna
4. notnull

Using the .isnull Function

We can check which cell null values are present using this function. An example is shown below.

Code

```
# Python program to check if a cell contains a null value or not

# Importing the required libraries
import pandas as pd
import numpy as np

# Creating a dataframe
data = {'numbers': [3, 6, 9, 2, 6, 8, np.nan, 1, 3, np.nan, 8, 9, 10, np.nan]}
dframe = pd.DataFrame(data)

# Checking the null values
print(dframe.isnull())
```

Output:

```
      numbers
0      False
1      False
2      False
3      False
4      False
5      False
6       True
7      False
8      False
9       True
10     False
11     False
12     False
13     True
```

Using the isnan() Method

Using the pandas.isnull function, we verified the NaN entries in the instance above. We will now employ a different technique termed isnan. This function is an in-built function of the numpy library. The code below shows an example of how to check the nan value in a particular cell.

Code

```
# Python program to check for the nan value in a cell using isnan function of numpy
# Importing the required libraries
import numpy as np
import pandas as pd
# Creating a data frame
data = {'numbers1': [3, 6, 9, 2, 6, 8, np.nan, 1, 3, np.nan, 8, 9, 10, np.nan, 8],
        'numbers2': [4, 6, np.nan, 5, 8, np.nan, 1, 6, np.nan, np.nan, 17, np.nan, 19, 7, 3]}
dframe = pd.DataFrame(data)
value = dframe.at[6, 'numbers1'] #nan
nan = np.isnan(value)
print("The particular value of the data frame is nan:", nan)
```

Output:

```
The particular value of the data frame is nan: True
```

Using .isna Function of Pandas

We will show how to use .isna function of Pandas.

Code

```
# Python program to check for nan values using .isna function

# Importing the required libraries
import numpy as np
import pandas as pd

# Creating a data frame
data = {'numbers1': [3, 6, 9, 2, 6, 8, np.nan, 1, 3, np.nan, 8, 9, 10, np.nan, 8],
        'numbers2': [4, 6, np.nan, 5, 8, np.nan, 1, 6, np.nan, np.nan, 17, np.nan, 19, 7, 3]}
df = pd.DataFrame(data)

print("checking for NaN value in the series numbers2 at the 6th index")
pd.isna(df.iloc[6,0])
```

Output:

```
checking for NaN value in the series numbers2 at the 6th index
True
```

Using .notnull Method

The notnull function is another way to determine whether a cell is NaN or a dataframe contains a null value. According to the program below, this function will give a boolean result of False if the given cell value is NaN or null.


```

# Importing the required libraries
import numpy as np
import pandas as pd

# Creating a data frame
data = {'numbers1': [3, 6, 9, 2, 6, 8, np.nan, 1, 3, np.nan, 8, 9, 10, np.nan, 8],
        'numbers2': [4, 6, np.nan, 5, 8, np.nan, 1, 6, np.nan, np.nan, 17, np.nan, 19, 7, 3]}
dframe = pd.DataFrame(data)

# checking NaN value in the cell, series and entire dataframe
print("The cell does not has a null value: ", pd.notnull(dframe.iloc[6,0]))
print("The series does has not null value: ", pd.notnull(dframe['numbers2']).values.any())
print("Number of no null value in the series of the dataframe: ", pd.notnull(dframe).sum())

```

Output:

```

The cell does not has a null value:  False
The series does has not null value:  True
Number of no null value in the series of the dataframe:  numbers1    12
numbers2    10

```

Hierarchical Indexing and Leveling in Pandas

Hierarchical indexing

Hierarchical Indexes are also known as multi-indexing is setting more than one column name as the index. we are going to use homelessness.csv file.

Example

```

# importing pandas library as alias pd
import pandas as pd

# calling the pandas read_csv() function.
# and storing the result in DataFrame df

```

```
df = pd.read_csv('homelessness.csv')
```

```
print(df.head())
```

Output

```
   region      state  individuals  family_members  state_pop
0  East South Central  Alabama      2570.0         864.0    4887681
1    Pacific      Alaska      1434.0         582.0     735139
2    Mountain    Arizona      7259.0        2606.0    7158024
3  West South Central  Arkansas      2280.0         432.0    3009733
4    Pacific  California    109008.0        20964.0   39461588
```

Reading and Writing Data: CSV or Text File

One of the most striking features of Pandas is its ability to read and write various types of files including CSV and Excel. You can effectively and easily manipulate CSV files in Pandas using functions

like `read_csv()` and `to_csv()`.

Installing Pandas

We have to install Pandas before using it. Let's use `pip`:

```
$ pip install pandas
```

Reading CSV Files with `read_csv()`

Let's import the Titanic Dataset, which can be obtained on [GitHub](#):

```
import pandas as pd
titanic_data = pd.read_csv('titanic.csv')
```

Pandas will search for this file in the directory of the script, naturally, and we just supply the file path to the file we'd like to parse as the one and only required argument of this method.

Let's take a look at the `head()` of this dataset to make sure it's imported correctly:

```
titanic_data.head()
```

This results in:

```
PassengerId  Survived  Pclass  ...  Fare  Cabin  Embarked
0            1         0       3  ...   7.2500   NaN        S
1            2         1       1  ...  71.2833   C85        C
2            3         1       3  ...   7.9250   NaN        S
3            4         1       1  ...  53.1000  C123        S
4            5         0       3  ...   8.0500   NaN        S
```

Alternatively, you can also read CSV files from online resources, such as GitHub, simply by passing in the URL of the resource to the `read_csv()` function. Let's read this same CSV file from the GitHub repository, without downloading it to our local machine first:

```
import pandas as pd

titanic_data = pd.read_csv(r'https://raw.githubusercontent.com/datasciencedojo/dataset')
print(titanic_data.head())
```

This also results in:

```
PassengerId  Survived  Pclass  ...   Fare Cabin  Embarked
0            1         0      3 ...   7.2500  NaN      S
1            2         1      1 ...  71.2833  C85      C
2            3         1      3 ...   7.9250  NaN      S
3            4         1      1 ...  53.1000  C123     S
4            5         0      3 ...   8.0500  NaN      S
```

```
[5 rows x 12 columns]
```

Writing CSV Files with `to_csv()`

Again, `DataFrame`s are tabular. Turning a `DataFrame` into a CSV file is as simple as turning a CSV file into a `DataFrame` - we call the `write_csv()` function on the `DataFrame` instance.

When writing a `DataFrame` to a CSV file, you can also change the column names, using the `columns` argument, or specify a delimiter via the `sep` argument. If you don't specify either of these, you'll end up with a standard Comma-Separated Value file.

Let's play around with this:

```
import pandas as pd
cities = pd.DataFrame([[ 'Sacramento', 'California'], [ 'Miami', 'Florida']], columns=[ '
cities.to_csv('cities.csv')
```

Here, we've made a simple `DataFrame` with two cities and their respective states. Then, we went ahead and saved that data into a CSV file using `to_csv()` and provided the filename.

This results in a new file in the working directory of the script you're running, which contains:

```
,City,State
0,Sacramento,California
1,Miami,Florida
```

Though, this isn't really well-formatted. We've still got the indices from the `DataFrame`, which also puts a weird missing spot before the column names. If we re-imported this CSV back into a `DataFrame`, it'd be a mess:

```
df = pd.read_csv('cities.csv')
print(df)
```

This results in:

```
Unnamed: 0      City      State
0          0  Sacramento  California
1          1      Miami    Florida
```

Pandas to read HTML from a string

Before you use Pandas to read HTML from a string, you need to install Pandas using **conda** or **pip** commands.

pip3 install pandas

conda install pandas

Once you've done that, you can create a Python file. Paste a line of code in which any variable contains HTML.

According to the [Pandas official documentation](#), the string can represent the HTML or a URL.

If you're using lxml, it will only accept the following protocols:

- Ftp

- File url
- Http

Therefore, if you're using a URL that begins with "HTTPS," remove the 's' from the end. After pasting the code with HTML, you can run the **read_html** function.

```
import pandas as pd  
df_list = pd.read_html(html)
```

The function will extract the data from HTML tables, showing you the list of tables. If you know the number of tables in the string, you can confirm that Pandas has read all of the DataFrames by using the following command:

```
print(len(df_list))
```

OUTPUT: 1

If your string only has one table, the **df_list** variable will confirm it. Finally, if you want to see the contents of the table in your string, you can use this command:

```
print(df_list[0])
```

It will extract the data from the HTML table/s and show it.

How to read HTML in Pandas through a URL

Pandas `read_html()` can also accept a URL. You can read HTML tables from websites directly into a pandas DataFrame by passing the URL to the `read_html()` function.

The function will return one DataFrame for each table on the page. In this Pandas read HTML example, the following URL is used: <https://int.soccerway.com/teams/rankings/fifa/>

To list the DataFrames, paste the following command:

```
dfs = pd.read_html(URL)
```

You will see a list of DataFrames. You can now type **len(dfs)** to see all the tables in the URL. The Pandas read HTML example URL has 1 table.

How to read HTML in Pandas through a file

You can also read HTML tables from a local file by passing the file path to the `read_html()` function. Suppose you have saved an HTML file called "table.html" in your working directory. The file path would be:

```
file_path = 'table.html'
```

Now, to read this table into a pandas DataFrame, run the following code:

```
file_path = "table.html"
with open(file_path, 'r') as f:
    dfs = pd.read_html(f.read())
dfs[0]
```

Reading Excel File using Pandas in Python

- Installing and Importing Pandas
- Reading multiple Excel sheets using Pandas
- Application of different Pandas functions

Reading Excel File using Pandas in Python

Installing Pandas

To install Pandas in Python, we can use the following command in the command prompt:

```
pip install pandas
```

To install Pandas in Anaconda, we can use the following command in Anaconda Terminal:

```
conda install pandas
```

Importing pandas

First of all, we need to import the Pandas module which can be done by running the command:

```
import pandas as pd
```

Now we can import the Excel file using the read_excel function in Pandas. The second statement reads the data from Excel and stores it into a pandas Data Frame which is represented by the variable newData.

```
df = pd.read_excel('Example.xlsx')

print(df)
```

Output:

	Roll No.	English	Maths	Science
0	1	19	13	17
1	2	14	20	18
2	3	15	18	19
3	4	13	14	14
4	5	17	16	20
5	6	19	13	17
6	7	14	20	18
7	8	15	18	19
8	9	13	14	14
9	10	17	16	20

Unit V

Programming with R

Introduction

- R is a popular programming language used for statistical computing and graphical presentation.
- Its most common use is to analyze and visualize data.
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has a large community support
- It has many packages (libraries of functions) that can be used to solve different problems.

R Variables

A variable is a memory allocated for the storage of specific data and the name associated with the variable is used to work around this reserved block.

Syntax

Using equal to operators

```
variable_name = value
```

using leftward operator

```
variable_name <- value
```

using rightward operator

```
value -> variable_name
```

3

Example

```
# using equal to operator  
var1 = "hello"  
  
print(var1)  
  
# using leftward operator  
var2 <- "hello"  
  
print(var2)  
  
# using rightward operator  
"hello" -> var3  
  
print(var3)
```

Output

```
[1] "hello"
```

```
[1] "hello"
```

```
[1] "hello"
```

Data Structures in R

The most essential data structures used in R include:

- Vectors
- Lists
- Dataframes
- Matrices
- Arrays
- Factors

Vectors

Vector is one of the basic data structures in R. It is homogenous, which means that it only contains elements of the same data type. Data types can be numeric, integer, character, complex, or logical.

Example

```
# Vectors(ordered collection of same data type)
```

```
X = c(1, 3, 5, 7, 8)
```

```
# Printing those elements in console
```

```
print(X)
```

Output

```
[1] 1 3 5 7 8
```

7

Lists

A [list](#) is a non-homogeneous data structure, which implies that it can contain elements of different data types. It accepts numbers, characters, lists, and even matrices and functions inside it. It is created by using the `list()` function.

Example

```
empId = c(1, 2, 3, 4)
```

```
empName = c("Debi", "Sandeep", "Subham", "Shiba")
```

```
numberOfEmp = 4
```

```
empList = list(empId, empName, numberOfEmp)
```

```
print(empList)
```

Output

```
[[1]] [1] 1 2 3 4
```

```
[[2]] [1] "Debi" "Sandeep" "Subham" "Shiba" [[3]]
```

Matrices

A matrix is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically. Matrices are two-dimensional, homogeneous data structures.

Example

```
M1 <- matrix(c(1:9), nrow = 3, ncol = 3, byrow = TRUE)
print(M1)
```

Output

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

9

Arrays

Arrays refer to the type of data structure that is used to store multiple items of a similar type together. This leads to a collection of items that are stored at contiguous memory locations. This memory location is denoted by the array name. The position of an element can be calculated simply by adding an offset to its base value.

Example

Array Structure

An array consists of the following:

Array Index: The array index identifies the location of the element. The array index starts with 0.

Array Element: Array elements are items that are stored in the array.

Array Length: The array length is determined by the number of elements that can be stored by the array. .

10