



KAH

Know About Hardware's

Syllabus

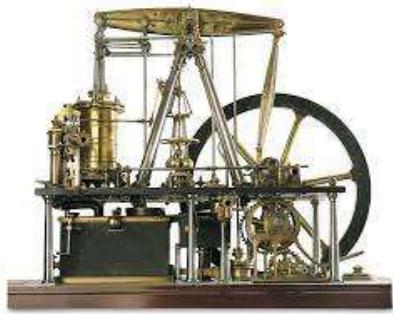


- History of Electronics
- History of computer
- Generation of Computers
- KAH Keyboard & mouse
- KAH Monitor
- KAH Processor
- KAH Mother board
- KAH HDD & SSD
- KAH Ram
- KAH OS

History of Electronics



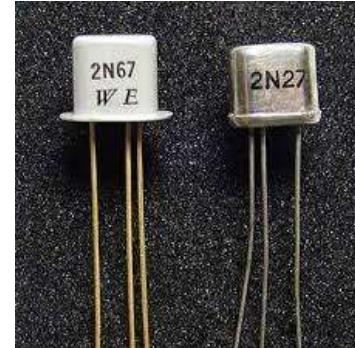
Steam Engine:



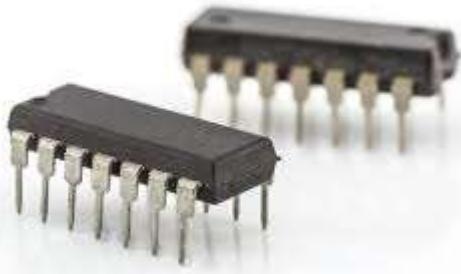
Vacuum Tubes:



Transistor:



IC(Integrated Circuit):



MicroProcessor:

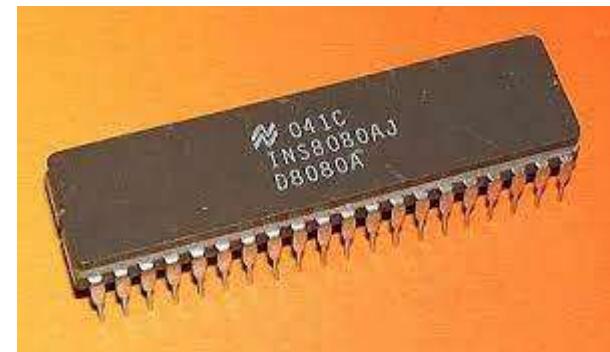
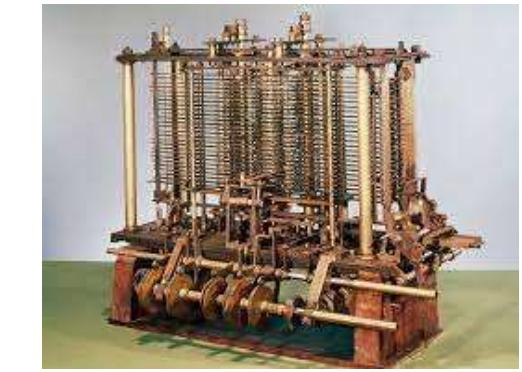


History of computers



Computer History:

- The first mechanical computer name is **Analytical Engine**.
- It was developed by **Charles Babbage** in **1833**.
- The first computer programmer was **Ada Byron** in **1840**.
- **MITS(Micro Instrumentation and Telemetry System)** made the first personal computer **Altair** in **1974**.
- The processor used in first personal computer was **Intel 8080**, It was developed in **1974**.



Generation of Computers



First Generation Computer:

- Main Electronic component – Vacuum Tube.
- Main Memory – Magnetic Drums and Magnetic Tapes.
- Programming Language – Machine Language.
- Power – Consume lot of energy and produce lot of heat.
- Speed – Very slow in speed.
- I/O devices – Punched Cards and Paper Tape.
- Example – ENIAC, IBM 650 etc....



Generations of Computers



Second Generation:

- **Main Electronic Component** – Transistor.
- **Memory** – Magnetic core and Magnetic tape/disk.
- **Programming Language** – Assembly Language.
- **Power** – Low Power consumption.
- **Speed** – Improvement of speed and Reliability.
- **I/O Devices** – Punched cards and Magnetic Tape.
- **Example** – IBM 1401, IBM 7090 etc...



Generations of Computers



Third Generation:

- **Main Electronic Component** – IC(Integrated Circuit).
- **Memory** – Large Magnetic core, Magnetic tape/disk.
- **Programming Language** – High Level Language(FORTRAN, Pascal, C etc...).
- **Size** – Smaller and Cheaper (They were called “minicomputers”).
- **Speed** – Improvement of speed and Reliability.
- **I/O Devices** – Magnetic Tape, Keyboard, Monitor etc...
- **Example** – IBM 360, IBM 370, PDP-11 etc...



Generations of Computers



Fourth Generation:

- **Main Electronic Component** – VLSI(Very Large Scale Integeration)Micro Processor.
- **Memory** – Semiconductor Memory (RAM, ROM etc...)
- **Programming Language** – High Level Language(Python, JAVA, C# etc...).
- **Size** – Smaller and Cheaper more efficient than Third generation.
- **Speed** – Improvement of speed, accuracy and Reliability.
- **I/O Devices** – Keyboard, pointing devices, printer Monitor etc...
- **Example** – IBM pc, STAR 1000, Apple Macintoss etc...



Generations of Computers



Fifth Generation:

- **Main Electronic Component** – ULSI(Ultra Large Scale Integeration)
Micro Processor.
- **Power** – Consumes less power and less heat.
- **Language** – Understandable natural language(Human Language).
- **Size** – Portable and small in size and have huge storage capacity.
- **Speed** – Improvement of speed, accuracy and Reliability.
- **I/O Devices** – Keyboard, Mouse, Trackpad, printer, Monitor etc...
- **Example** – Desktop, Laptop, Mobile Phones etc...



Fifth generation computers are in designing mode with Artificial Intelligence technology.



KAH Keyboard & mouse



Keyboard:

- Keyboard is the most **common and very popular** input device.
- The layout of the keyboard is like that of **traditional typewriter**.
- Keyboards are two sizes **84 keys or 101 keys**.



Mouse:

- Mouse is the most popular **pointing device**.
- It is a very famous **cursor-control device** having a **small ball** at its base.
- The ball senses the movement of the mouse and send the signals
To the CPU.



KAH Monitor



Monitor:

- A computer monitor is an **output device** that displays information's in **pictorial or textual form**.
- A computer monitor is also one of the important part of a computer system.

Types of monitors:

- CRT(Cathode Ray Tube)
- LCD(Liquid Crystal Display)
- LED(Light Emitting Diode)



CRT



LCD



LED

KAH Processor



processor:

- An computer processor is also Known as **CPU**(Central Processing Unit).
- It is considered as the **brain of the computer**.
- It performs all type of data processing operations.
- It stores data, intermediate results and Instructions.
- It controls the operation of all parts of the computer.
- Popular processor manufacturer's are **Intel, AMD**.



KAH Motherboard



Motherboard:

- The motherboard is the **backbone** of the computer.
- It ties all components together at one spot.
- Total motherboard functionality is necessary for a computer to work well.
- Popular motherboard manufacturers are **Intel , AMD, Gigabyte, MSI**.



HDD(HDD):

- HDD Stands for Hard Disk Drive.
- An hard disk is an magnetic storage medium for a computer.
- It is a flat circular plate made of aluminum or glass and coated with magnetic material.
- Hard Disk for personal computers can store terabytes of information.



KAH HDD & SSD



SSD(ROM):

- SSD Stands for **Solid State Drive**.
- It is **smaller and faster** than hard disk.
- It allows pc to be **thinner** and more **lightweight**.
- SSDs are most common storage drives today.
- SSD is **costlier** than Hard Disk.



KAH RAM



RAM:

- RAM stands for Random Access Memory.
- It is a **read/write memory** which stores data until the machine is working.
- It is also known as **temporary memory**.
- Ram is **volatile** in nature.
- Ram is small both in **physical size** and the **amount of data** it can hold.



Thank you



KAS

Know About Software

Syllabus



- Binary Language(0's & 1's)
- Instruction Sets
- Assembly Language
- OS
- UNIX OS(Assembly language)
- C Language
- UNIX OS (Using C)
- Function oriented
- Oops oriented

Binary Language



Binary Language:

- Binary Describes a **numbering scheme** in which there are only **two possible** values for each digit **0 or 1**.
- These system use this **code** to understand **operational instructions** and **user input** and to present relevant output to the **user**.

Character	ASCII code	Binary code
null character	0	0000000
a	97	1100001
b	98	1100010
c	99	1100011
A	65	1000001
B	66	1000010
C	67	1000011
%	37	0100101
+	43	0101011
0	48	0110000
1	49	0110001
Delete	127	1111111

Instruction Set



Instruction Set:

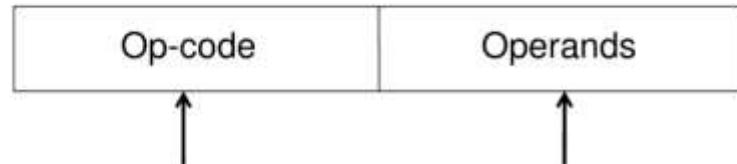
- An **Instruction set** is a group of commands for **CPU**.
- It defines how the CPU is controlled by the **Software**.
- The **ISA(Instruction Set Architecture)** acts as an interface between the hardware and the software.

Instruction Set



Instruction Format

Format of a typical microprocessor instruction



Identifies the action
to be taken

Identifies the data to
be operated

E.g. MOV AX,BX

ADD AX,BX

•Op-code are usually written in the form called mnemonic.

E.g. MOVE → MOV

ADDITION → ADD

INCREASE → INC

Assembly Language



Assembly Language:

- It is an **low level programming language** that is intended to communicate directly with **Computer's Hardware**.
- Unlike **Machine Language**, **Assembly Language's** are designed to be readable by **Human's**.

```
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
; nasm -felf64 hello.asm && ld hello.o && ./a.out
;

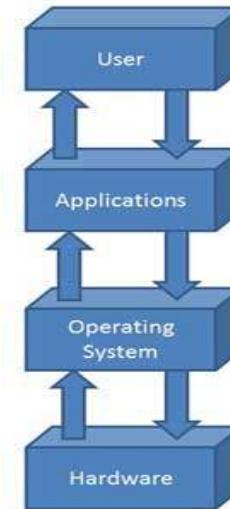
    global  _start

    section .text
_start:  mov     rax, 1          ; system call for write
        mov     rdi, 1          ; file handle 1 is stdout
        mov     rsi, message    ; address of string to output
        mov     rdx, 13         ; number of bytes
        syscall               ; invoke operating system to do the write
        mov     rax, 60          ; system call for exit
        xor     rdi, rdi        ; exit code 0
        syscall               ; invoke operating system to exit

    section .data
message: db      "Hello, World", 10      ; note the newline at the end
```

OS:

- OS Stands for **Operating System**.
- An Operating System is a program that acts as an **interface** between the **software** and **hardware**.
- It is an Integrated set of **specialized programs** used to manage operations of the computer.
- The first OS used for real work is **GM-NAA I/O** by **IBM**.

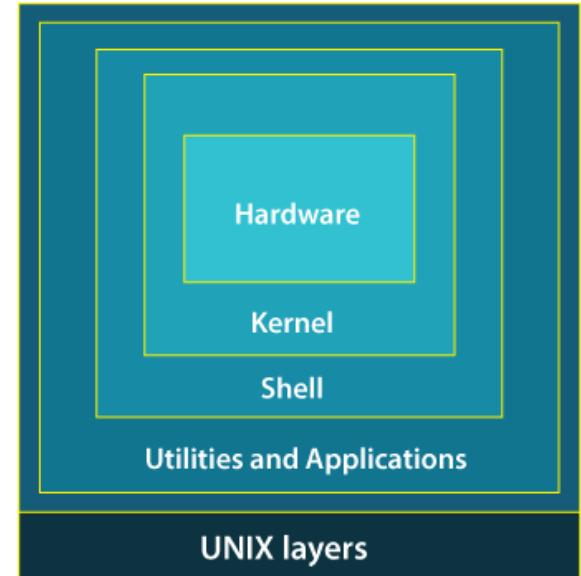


UNIX OS(Assembly Language)



UNIX Operating System:

- UNIX is a **Powerful** Operating System developed by **Dennis Ritchie** and **Ken Thompson**.
- It was developed in **AT&T Bell Laboratories** on 1970.
- It was written using **Assembly Language**.



C Language



C Language:

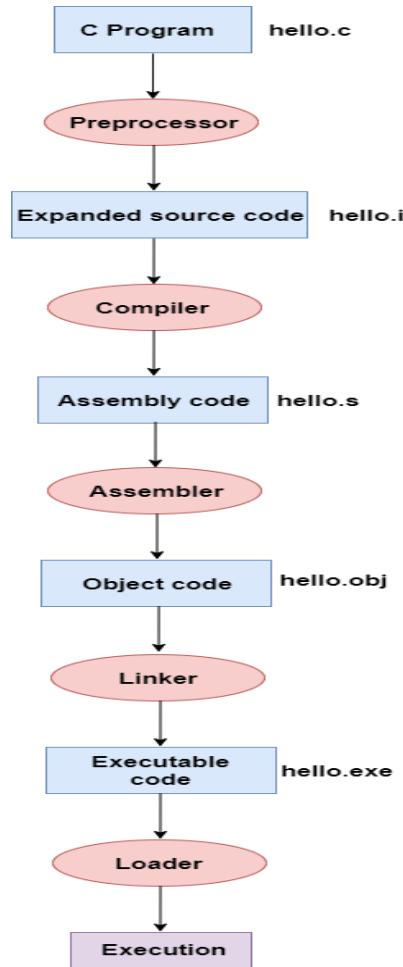
- C is an **General-Purpose** programming language.
- It was created by **Dennis Ritchie** at the **Bell Laboratories** in **1972**.
- It is **Function Oriented**.
- It was developed to write **UNIX OS**.



C language



C Programming work flow diagram:



UNIX OS(using C)



UNIX OS(using C):

- UNIX OS Code was rewritten using **C Programing**.
- It was rewritten in **1974**.
- It was rewritten because The code was more **compact** and it was easier to port to other processors.

Function Oriented



Function Oriented:

- Functional programming is a programming technique that requires **functional factors** required for creating and implementing the programs.
- The programming model used in functional programming is **declarative programming model**.
- In functional programming **variables and functions** are the main elements of the code.
- **C programing** is Function oriented.

Object Oriented



Object Oriented:

- Oops(Object oriented program) is an conceptual programming techniques that uses objects as the key.
- The Object oriented Programming uses the imperative programming model.
- In Object oriented programing the key elements are Objects and Methods.
- Java is Object oriented.

Example



Function:

```
#include<stdio.h>
int main(){
    for(int i = 1; i <= 10; i++){
        printf("%d\n", i);
    }
    return 0;
}
```

Oops:

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Thank You



About Python

About Python



Python Introduction:

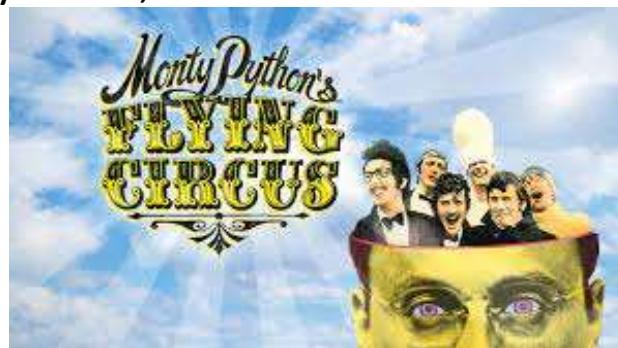
- Python is a widely used general-purpose, high level programming language.
- It was designed with an emphasis on code readability.
- Its syntax allows programmers to express their concepts in fewer lines of code.
- Python is maintained by Python Software Foundation a non-profit-membership organization and a community devoted to developing, improving, expanding and popularizing the python language and its environment.

About Python



Python History:

- Python was developed by *Guido Van Rossum* in *February 20 1991*.
- Many think the name python is used after the large snake python.
- Actually, The name of python programming language comes from an old BBC television comedy sketch series called "*Monty Python's Flying Circus*".
- One of the amazing features of Python is the fact that is actually one person's work.
- If you read the python documentations, you will see many examples that are inspired by the Monty Python comedy series, we can see mention of spam, eggs, lumberjack etc..



About Python



Why Python?

- *It is easy to learn:* The time needed to learn Python is shorter than for many other languages, This means that it is possible to start the actual programming faster.
- *It is easy to use for writing new software:* It's often possible to write code faster when using python.
- *It is easy to obtain, Install and deploy:* Python is free, open and multiplatform.

About Python



Install Python:

- Download Python installer from official website.



- Double click the Python Executable Installer.

Today (1)			
 python-3.11.3-amd64	5/6/2023 5:52 PM	Application	24,753 KB

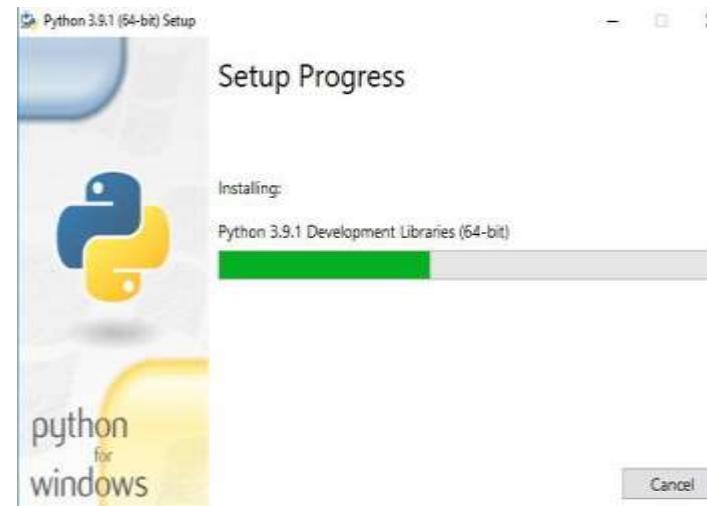
About Python



- After opening the installer, Click Install Now.



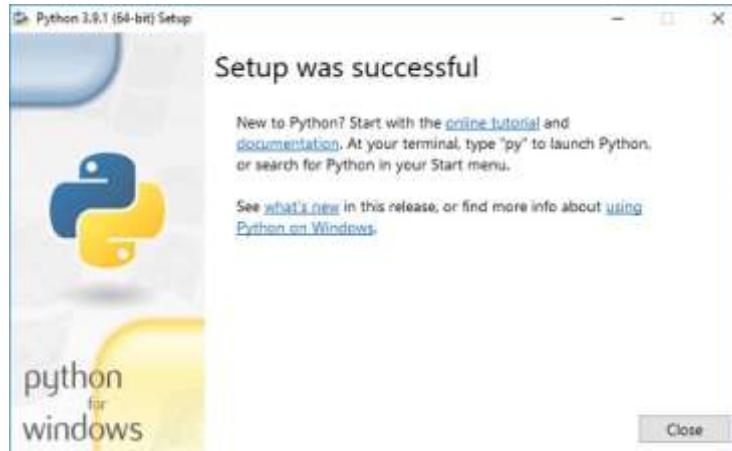
- Wait Until the installation is completed.



About Python



- Once the installation is successful, The following screen is displayed.



- To verify the python installation open command prompt and type “*python*”.

```
Command Prompt - python
Microsoft Windows [Version 10.0.17134.1304]
(c) 2018 Microsoft Corporation. All rights reserved.

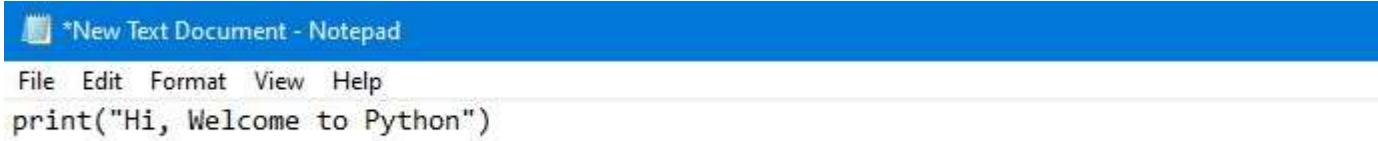
C:\Users\Inderjit Singh>python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

About Python



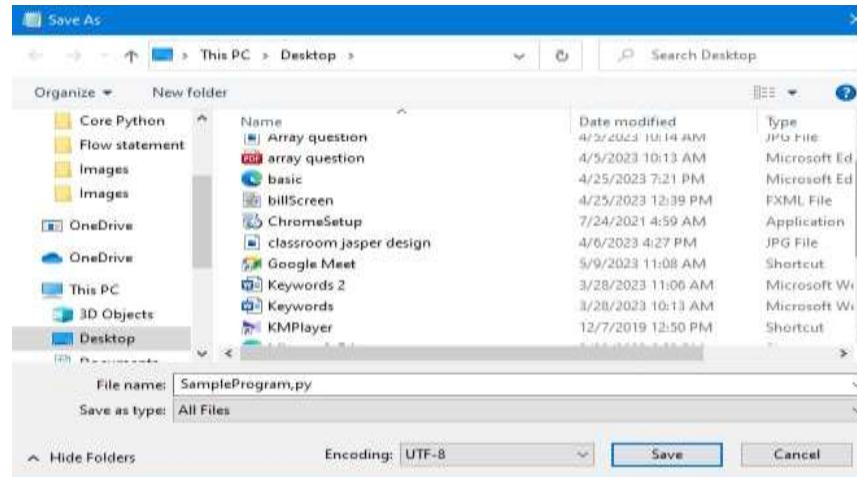
Writing a simple program using Notepad:

- Open your Text editor and write following code.



```
*New Text Document - Notepad
File Edit Format View Help
print("Hi, Welcome to Python")
```

- Now, save your file with following extension(.py) and save as type as (all files).



About Python



- An python file will be created on your destined location.



- Open command prompt and redirect it to your file location.

```
C:\Users\DELL>cd desktop  
C:\Users\DELL\Desktop>
```

- Now type python and after an space type the filename with .py extension.

```
C:\Users\DELL\Desktop>python sampleProgram.py  
Hi, Welcome to Python  
  
C:\Users\DELL\Desktop>
```

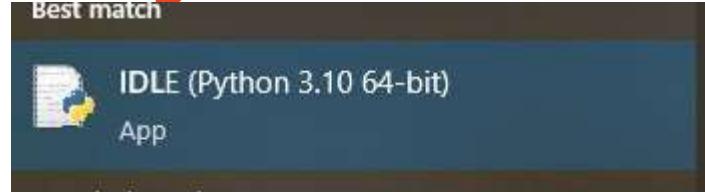
- We have successfully executed our first python program.

About Python

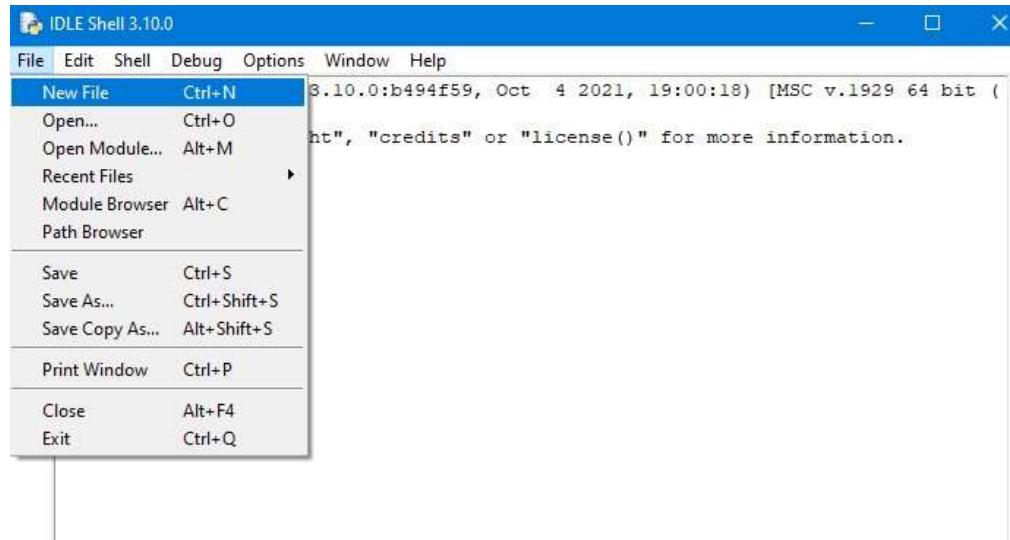


Writing a Simple Program using IDLE:

- Open IDLE python application.



- After opening the IDE select file and select new file.



About Python



- After opening new file write the following code.

```
print("Hello Developer")
print("Welcome to Python")
```

- Select save as from file menu and give your file a name.

This PC > New Volume (D:) > Syllabus > Python > Example			
Name	Date modified	Type	Size
simplePythonCode	5/6/2023 6:14 PM	Python Source File	1 KB

About Python



- Run the python file by pressing F5.

The screenshot shows the IDLE Shell 3.10.0 interface. The title bar reads "IDLE Shell 3.10.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python 3.10.0 startup message:

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: D:/Syllabus/Python/Example/simplePythonCode.py =====
Hello Developer
Welcome to Python
>>>
```

- Hurray! Our program has been executed successfully.

Thank You



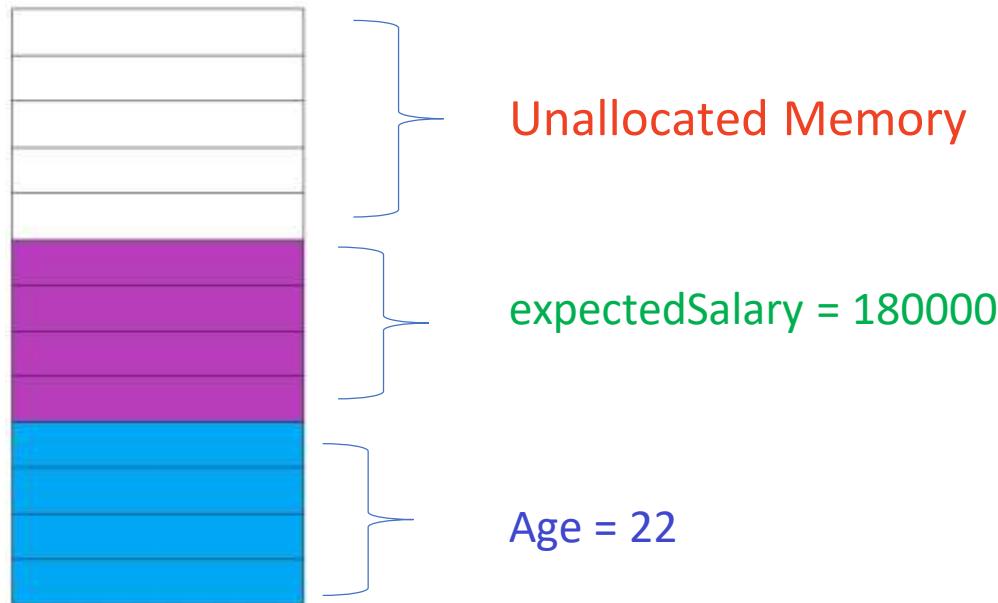
Variable & Datatype

Variables



Variables:

- A Variable is a **container** which holds the value while the program is being executed.
- Variable is a name of **Memory Location**.
- There are three types of Variables they are: **Local, Instance and Static**.





Variable

Syntax:

Declaring a variable:

```
variable name= value
```

Getting value from a variable:

```
print(variable_name)
```

Example:

Declaring a variable:

```
age = 21
```

Getting value from a variable:

```
print(age)
```

Datatypes



Datatypes:

- Datatypes specifies the different sizes and values that can be stored in a variable.

Inbuilt datatypes in python:

- Numeric types: int, float, complex.
- Text type: string.
- Sequence types: list, tuple, range, set, dict.
- Boolean type: Bool.

Datatypes



Numeric Datatypes:

Int :

The **int** datatype represents integer numbers.

Example: **a=20**

Float :

The **float** datatype represents floating point numbers.

Example: **a= 20.35**

Complex:

The **complex number** is a number that's been written with arithmetic functions.

Example: **a= 3+5j**

Datatypes



Text type:

String:

A **String** is represented by group of characters, **Strings** are enclosed with Single quotes('') & double quotes("") .

Example:

a= “ hello world”

Datatypes



Datatype Memory Size:

Type Code	Description	C Type	Size (in bytes)	Python C
bool	boolean	unsigned char	1	bool
int8	8-bit integer	signed char	1	int
uint8	8-bit unsigned integer	unsigned char	1	int
int16	16-bit integer	short	2	int
uint16	16-bit unsigned integer	unsigned short	2	int
int32	integer	int	4	int
uint32	unsigned integer	unsigned int	4	long
int64	64-bit integer	long long	8	long
uint64	unsigned 64-bit integer	unsigned long long	8	long
float16 [1]	half-precision float		2	
float32	single-precision float	float	4	float
float64	double-precision float	double	8	float
float96 [1] [2]	extended precision float		12	
float128 [1] [2]	extended precision float		16	
complex64	single-precision complex	struct {float r, i;}	8	complex
complex128	double-precision complex	struct {double r, i;}	16	complex
complex192 [1]	extended precision complex		24	
complex256 [1]	extended precision complex		32	
string	arbitrary length string	char[]		str
time32	integer time	POSIX's time_t	4	int
time64	floating point time	POSIX's struct timeval	8	float
enum	enumerated value	enum		

Datatypes



Sequence Datatypes:

List :

A **List** represents a group of elements written inside of square bracket([]).
List can store different types of elements.

Example:

a=[10,20.45,"Hello world"]

The above example will create an List with different types of elements for variable a.

Sequence Datatype methods



List Methods:

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
<u>sort()</u>	Sorts the list

Datatypes



Sequence Datatypes:

Tuple:

A **tuple** is similar to list but the elements in **tuple** cannot be modified.

Example:

```
a=(10,20.45,"Hello world")
```

The above example will create an Tuple with different types of elements for variable a.

Sequence Datatype methods



Tuple Methods:

Method	Description
<u>count()</u>	Returns the number of times a specified value occurs in a tuple
<u>index()</u>	Searches the tuple for a specified value and returns the position of where it was found

Datatypes



Sequence Datatypes:

Set:

A **set** is an unordered collection of elements, it means the elements may not appear in the same order as they are entered into the **set**. A set does not accept duplicate elements.

Example:

a={10,20,30,40,30}

Output:

(30,20,40,10)

In above example the output is printed in unordered format and duplicate values has been removed, hence it is an **set**.

Sequence Datatype methods



Set Methods:

Method	Description
<u>add()</u>	Adds an element to the set
<u>clear()</u>	Removes all the elements from the set
<u>copy()</u>	Returns a copy of the set
<u>difference()</u>	Returns a set containing the difference between two or more sets
<u>difference_update()</u>	Removes the items in this set that are also included in another, specified set
<u>discard()</u>	Remove the specified item
<u>intersection()</u>	Returns a set, that is the intersection of two or more sets
<u>intersection_update()</u>	Removes the items in this set that are not present in other, specified set(s)
<u>isdisjoint()</u>	Returns whether two sets have a intersection or not
<u>issubset()</u>	Returns whether another set contains this set or not
<u>issuperset()</u>	Returns whether this set contains another set or not
<u>pop()</u>	Removes an element from the set
<u>remove()</u>	Removes the specified element
<u>symmetric_difference()</u>	Returns a set with the symmetric differences of two sets
<u>symmetric_difference_update()</u>	inserts the symmetric differences from this set and another
<u>union()</u>	Return a set containing the union of sets
<u>update()</u>	Update the set with another set, or any other iterable

Datatypes



Sequence Datatypes:

Dictionary:

Dictionaries are used to store the data in **key: value** pairs, an dictionary does not allow duplicate value.

Example:

```
dict={ "model": "Mustang",
       "year": "1986",
       "brand": "Ford"}
```

In above example an variable has been declared with curly braces with keys and values, hence it is known as **Dictionary**.

Sequence Datatype methods



Dictionary Methods:

Method	Description
<u>clear()</u>	Removes all the elements from the dictionary
<u>copy()</u>	Returns a copy of the dictionary
<u>fromkeys()</u>	Returns a dictionary with the specified keys and value
<u>get()</u>	Returns the value of the specified key
<u>items()</u>	Returns a list containing a tuple for each key value pair
<u>keys()</u>	Returns a list containing the dictionary's keys
<u>pop()</u>	Removes the element with the specified key
<u>popitem()</u>	Removes the last inserted key-value pair
<u>setdefault()</u>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<u>update()</u>	Updates the dictionary with the specified key-value pairs
<u>values()</u>	Returns a list of all the values in the dictionary

Example 1



Code:

```
VariableAndDatatype.py - D:/Syllabus/Python/Example/VariableAndDatatype.py (3.10.0)
File Edit Format Run Options Window Help

name = "Steve Austin"
print(name)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: D:/Syllabus/Python/Example/VariableAndDatatype.py =====
Steve Austin
>>>
```

Example 2



Code:

```
VariableAndDatatype.py - D:/Syllabus/Python/Example/VariableAndDatatype.py (3.10.0)
File Edit Format Run Options Window Help

age = 23
print(age)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help

Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: D:/Syllabus/Python/Example/VariableAndDatatype.py =====
23
>>>
```

Example 3



Code:

```
a = 10
b = 5
c = a+b
print(c)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Syllabus/Python/Example/VariableAndDatatype.py =====
15
|
```

Example 4



Code:

```
VariableAndDatatype.py - D:/Syllabus/Python/Example/VariableAndDatatype.py (3.10.0) - □ X
File Edit Format Run Options Window Help

studentNames = ["Ram", "Tamil", "shreya"]
print(studentNames)
```

Output:

```
IDLE Shell 3.10.0 - □ X
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: D:/Syllabus/Python/Example/VariableAndDatatype.py =====
['Ram', 'Tamil', 'shreya']
>>>
```

Example 5



Code:

```
VariableAndDatatype.py - D:/Syllabus/Python/Example/VariableAndDatatype.py (3.10.0) - X
File Edit Format Run Options Window Help

studentNames = {"Ram", "Tamil", "Ram", "shreya"}
print(studentNames)
```

Output:

```
IDLE Shell 3.10.0 - X
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Syllabus/Python/Example/VariableAndDatatype.py ======
{'shreya', 'Tamil', 'Ram'}
>>>
```

Example 6



Code:

```
VariableAndDatatype.py - D:/Syllabus/Python/Example/VariableAndDatatype.py (3.10.0)
File Edit Format Run Options Window Help

studentDetails = {"Name": "Ragu",
                  "Age" : 25,
                  "Major" : "BCA"
                 }
print(studentDetails)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: D:/Syllabus/Python/Example/VariableAndDatatype.py =====
{'Name': 'Ragu', 'Age': 25, 'Major': 'BCA'}
>>>
```

Thank You



Operators

Operators



What is an Operator?

- An Operator is a character that represents a specific mathematical or logical action or process.
- The values that an operator acts on are called operands.
- Some important operators programmers use :
 - Arithmetic operator
 - Assignment operator
 - Logical operator

Operators



Types Of Operators:

- Arithmetic Operator
- Assignment Operator
- Comparison Operator
- Logical Operator
- Identity Operator
- Membership Operator

Operators



Arithmetic operator:

Arithmetic operators are used with numeric values to perform common mathematical operations.

Operators



Arithmetic operator:

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$++x$
--	Decrement	Decreases the value of a variable by 1	$--x$

Operators



Example:

```
operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)

File Edit Format Run Options Window Help
a = 10
b = 5

print("Addition: ", (a+b))
print("Subtraction: ", (a-b))
print("Miltiplication: ", (a*b))
print("Divition: ", (a/b))
print("Modulus: ", (a%b))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
Addition: 15
Subtraction: 5
Miltiplication: 50
Divition: 2.0
Modulus: 0
>>>
```

Operators



Assignment operator:

- **Assignment operator** is used to assign the value, variable and function to another variable.

Operators



Assignment operator:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

Operators



Example:

```
*operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)*
File Edit Format Run Options Window Help
#assign value using =
a = 13
var = 0

var +=a
print("var using += ", var)

var *=a
print("var using *= ", var)

var -=a
print("var using -= ", var)

var /=a
print("var using /= ", var)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
var using += 13
var using *= 169
var using -= 156
var using /= 12.0
>>>
```

Operators



Comparison operator:

- Comparison operators are used to Compare two values.
- If the condition is true it returns true. Else, it returns false.

Operators



Comparison operator:

Operator	Name	Example
<code>==</code>	Equal to	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

Operators



Example:

```
*operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)*
File Edit Format Run Options Window Help
a = 10
b = 20
equal = a==b
notEqual = a!=b
greater = a>b
lesser = a<b
greaterEqual = a>=b
lesserEqual = a<=b

print("The value in a is equal to b : ", equal)
print("The value in a is not equal to b : ", notEqual)
print("The value in a is greater than b : ", greater)
print("The value in a is lesser than b: ", lesser)
print("The value in a is greater/equal with b : ", greaterEqual)
print("The value in a is lesser/equal with b : ", lesserEqual)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
The value in a is equal to b : False
The value in a is not equal to b : True
The value in a is greater than b : False
The value in a is lesser than b: True
The value in a is greater/equal with b : False
The value in a is lesser/equal with b : True
>>>
```

Operators



Logical operator:

- **Logical operators** are used on conditional statements.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

Operators



Logical AND operator:

- Logical and operator returns True if both the operands are True else it returns False.

Example:

```
(operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)*
File Edit Format Run Options Window Help
a = 10
b = 5
c = 15
d = a>b and b<c

print("The returned value after condition: ", d)

e = a>b and b>c
print("The returned value after wrong condition: ", e)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
The returned value after condition: True
The returned value after wrong condition: False
```

Operators



Logical OR operator:

- Logical or operator returns True if either of the operands are True, else it returns False.

Example:

```
operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)

File Edit Format Run Options Window Help

a = 10
b = 5
c = 15
d = a>b or b>c

print("The returned value after one condition is wrong: ", d)

e = a<b and b>c
print("The returned value after both condition wrong: ", e)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help

Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
The returned value after one condition is wrong: True
The returned value after both condition wrong: False
```

Operators



Logical NOT operator:

- Logical not operator works with single boolean value.
- If the boolean value is True it returns False and vice-versa.

Example:

```
operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)

File Edit Format Run Options Window Help
a = 10
b = 5
c = 15
d = a>b or b>c

print("The returned value after one condition is wrong: ", d)
print("The returned value after using not : ", not d)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
The returned value after one condition is wrong: True
The returned value after using not : False
>>> |
```

Operators



Identity operator:

- **Identity operators** are used to compare two object whether the objects are same and share same memory location.

Operator	Syntax	Description
is	x is y	This returns True if both variables are the same object
is not	x is not y	This returns True if both variables are not the same object

Operators



Identity is operator:

- Identity is operator evaluates to True if the variables on either side of the operator point to the same objects. else, false.

Example:

```
operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)

File Edit Format Run Options Window Help
bikel = ["RX-100", "R15", "BMW"]
bike2 = bikel

print("bikel is equal to bike 2:", bikel is bike2)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
bikel is equal to bike 2: True
>>>
```

Operators



Identity is not operator:

- Identity is not operator evaluates to True if both variables are not same the same objects. Else, False.

Example:

```
operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)
File Edit Format Run Options Window Help
bikel = ["RX-100", "R15", "BMW"]
bike2 = bikel

print("bikel is equal to bike 2:", bikel is not bike2)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/operator.py =====
bikel is equal to bike 2: False
>>>
```

Operators



Membership operator:

- **Membership operators** are used to test if a sequence is presented in an object.

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

Operators



Membership in operator:

- The in operator is used to check if a character exists in a sequence or not.

Example:

```
*operator.py - D:/Syllabus/Python/Example/operator.py (3.10.0)*
File Edit Format Run Options Window Help
brands = ["Apple", "Samsung", "Oneplus", "Redmagic"]
selected = "Apple"

print("check whether Selected brand is available:", selected in brands)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Syllabus/Python/Example/operator.py =====
check whether Selected brand is available: True
>>>
```

Operators



Membership not in operator:

- The not in operator evaluates to true if it does not find a variable in the specified sequence and false otherwise.

Example:

```
operator.py - D:\Syllabus\Python\Example\operator.py [3.10.0] - X
File Edit Format Run Options Window Help
brands = ["Apple", "Samsung", "Oneplus", "Redmagic"]
selected = "VIVO"

print("check whether Selected brand is not available:", selected not in brands)
```

Output:

```
IDLE Shell 3.10.0 - X
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:\Syllabus\Python\Example\operator.py =====
check whether Selected brand is not available: True
>>>
```

Thank You



Flow Statement

Flow Statement



Flow Statement:

- Control flow is the order in which the program's code executes.
- The control flow of a python program is regulated by conditional statements, loops and function calls.

Three types of control structures:

- ✓ Sequential statement
- ✓ Conditional statement
- ✓ Looping statement

Flow Statement



Sequential statement:

- Sequential statements are a set of statements whose execution process happens in a sequence.
- The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.
- It is the default mode of execution.

Flow Statement



Example:

```
sequentialStatement.py - D:/Syllabus/Python/Example/sequentialStatement.py (3.10.0)
File Edit Format Run Options Window Help
# Python program to calculate the area of a triangle.
x = 4
y = 5
z = 6
# Calculating the semi-perimeter.
s = (x + y + z) / 2
# Calculating the area of triangle.
area = (s * (s -x) * (s -y) * (s - z)) ** 0.5
print('Area of triangle = %0.2f'%area)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit | AMD64] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Syllabus/Python/Example/sequentialStatement.py ======
Area of triangle = 9.92
>>>
```

Flow Statement



Conditional Statement:

- A conditional statement is also known as selection control statement and decision control statement.
- It is based on the condition or decision.
- If the condition is true, then the block of statement will be executed.
- If the condition is false, then the block of statements will not execute.

Three conditional statement:

- ✓ If statement
- ✓ If – else statement
- ✓ If – elif – ladder statement
- ✓ Nested if statement

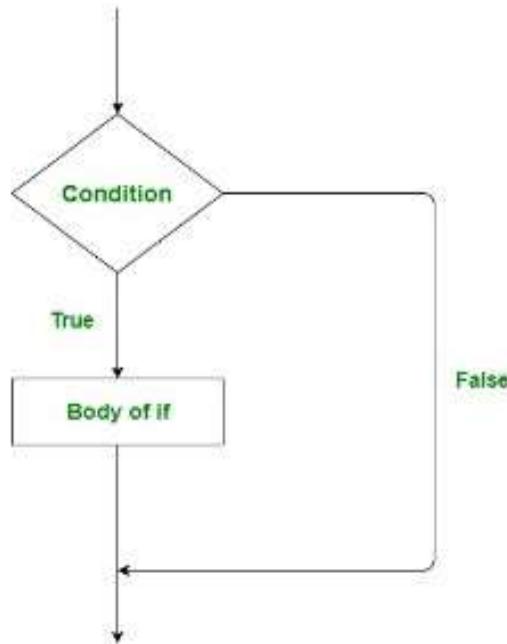
Flow Statement



If statement:

- If statement is used to run a set of codes only when the given condition is true.

If statement work flow:



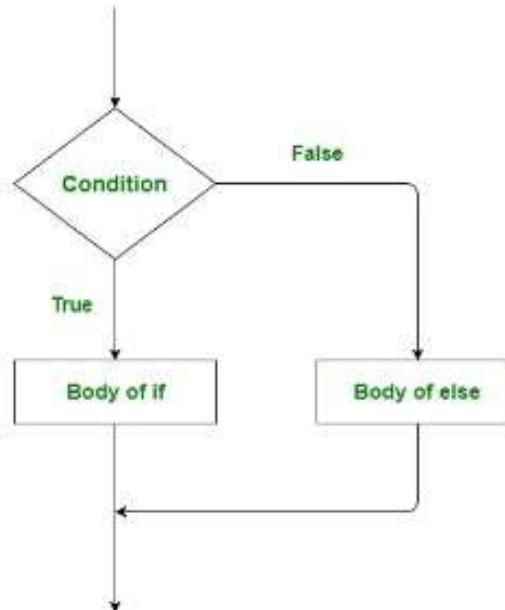
Flow Statement



If – else statement:

- If...else statement executes a block of codes if an condition is True, if the condition is false another block of codes will be executed.

If – else statement work flow:



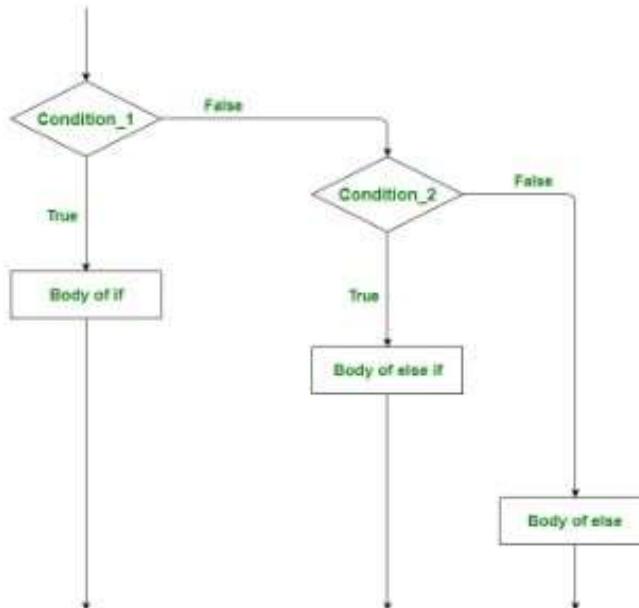
Flow Statement



If – elif – else ladder:

- If...elif...else statement helps us to make choices more than two alternatives.

If – elif – else ladder:



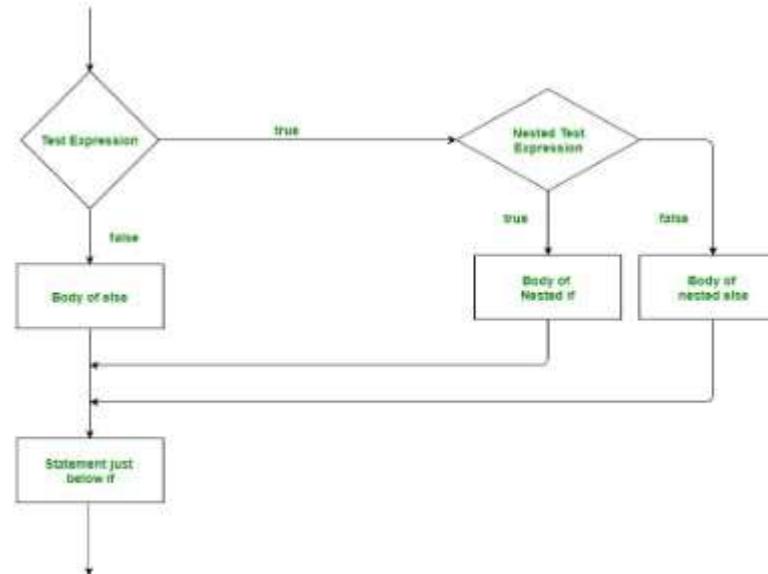
Flow Statement



Nested if statement:

- The Nested If Statements are the nesting of an if statement inside another if statement inside another if statement with or without an else statement.

Nested if statement work flow:



Flow statement



Looping Statement:

- A loop control statement allows us to execute or repeat a statement or a group of code multiple times.
- As long as the test condition is true, a block of code is repeated again and again.

Types of Loop control statement:

- For loop
- While loop

Flow Statement



For loop:

- For loop is a counting loop that repeats a block of statements a certain number of times.

For loop work flow:

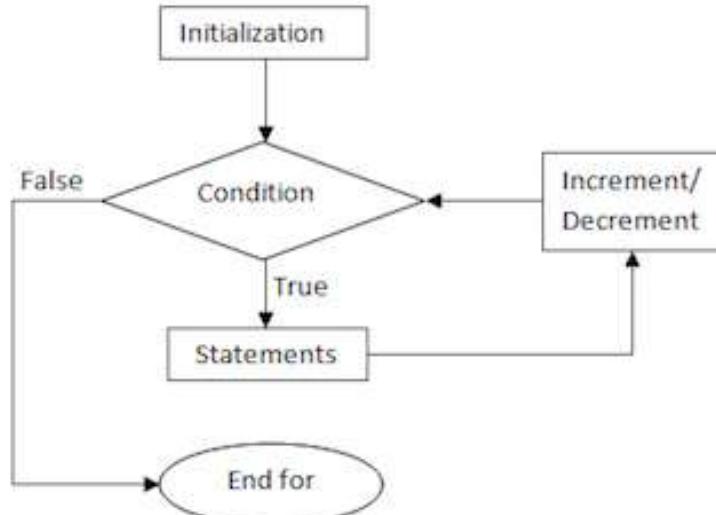


fig: Flowchart for for loop

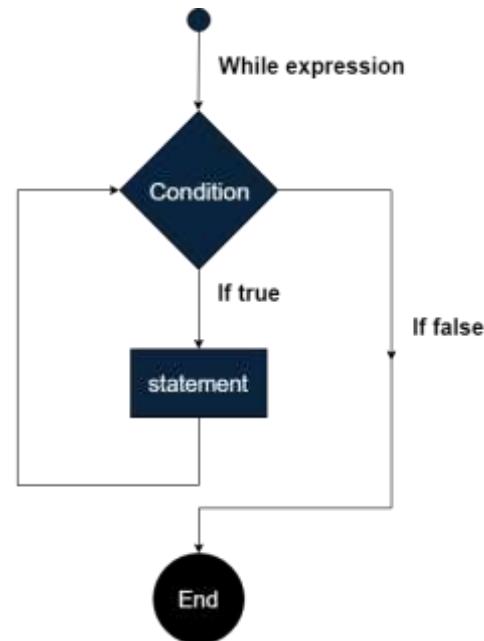
Flow Diagram



While Loop:

- While loop is a conditional loop that keeps repeating a set of statements as long as some test condition is true.

While Loop work flow:



Thank You



Loops in Python

Loop



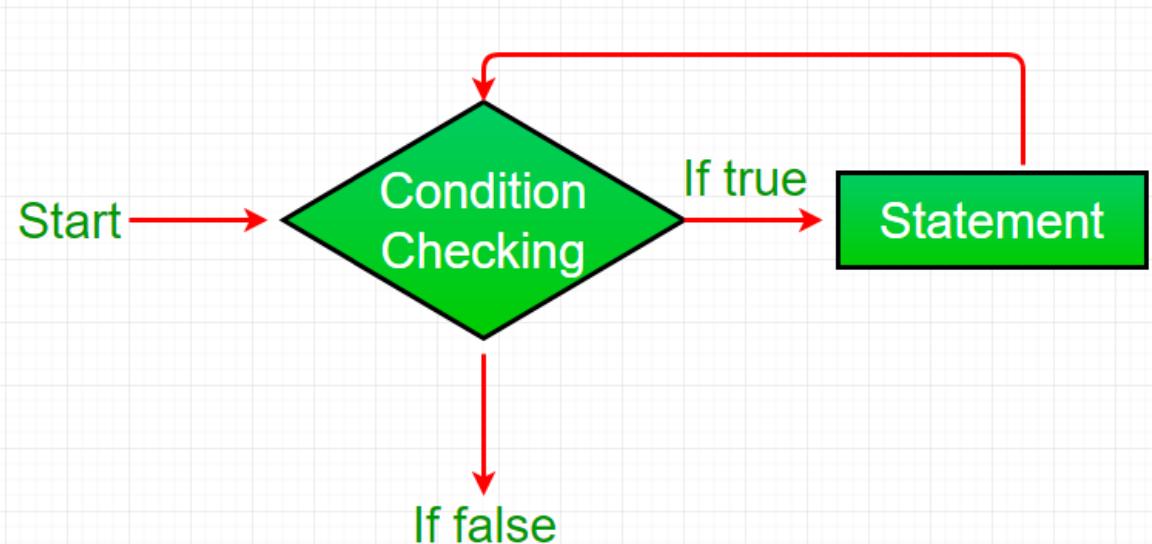
What is a loop ?

Loop is a feature used to execute a particular part of a program repeatedly if a given condition evaluates to be True.

Types of Loop:

- For Loop
- While Loop
- Do – While Loop

Loop work flow:



Loop



For Loop:

- The **for loop** is used to iterate a given set of statements **multiple times**.
- The for loop functions by running a **section of code repeatedly** until a certain condition has been satisfied.

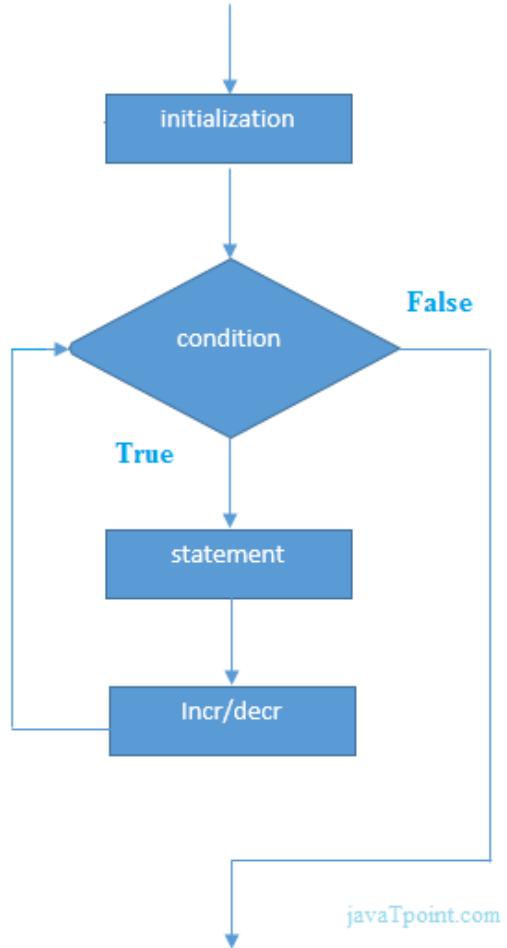
Syntax of For loop:

```
for iterator_variable in sequence:  
    statements
```

Loop



Flow chart:



Loop



Example:

```
numbers = [1, 3, 5, 7, 9, 11]
for num in numbers:
    sum_ = num ** 2
    print("Square root of ", num, " is ", sum_)
```

Output:

The screenshot shows the Python IDLE Shell 3.10.0 interface. The title bar reads "IDLE Shell 3.10.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python interpreter's prompt (>>>) followed by the code and its output. The output shows the square of each number from 1 to 11.

```
>>> Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: D:/Syllabus/Python/Example/loop.py =====
Square root of  1  is  1
Square root of  3  is  9
Square root of  5  is  25
Square root of  7  is  49
Square root of  9  is  81
Square root of  11  is  121
```

Loop



range() function:

- The range() function returns a sequence of numbers, starting from 0 by default and increments by 1 by default and stops before a specified number.

Syntax:

range(start, stop, step)

Parameter values:

- *Start:* Optional. An integer number specifying at which position to start. Default is 0.
- *Stop:* Required. An integer number specifying at which position to stop.
- *Step:* Optional. An integer number specifying the increments. Default is 1.

Loop



Example:

loop.py - D:/Syllabus/Python/Example/loop.py (3.10.0)

File Edit Format Run Options Window Help

```
n = 10

for i in range(0, n):
    print(i)
```

Output:

IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Syllabus/Python/Example/loop.py =====
0
1
2
3
4
5
6
7
8
9
>>>
```

Loop



Sequence datatypes in for loop:

Example:

```
loop.py - D:/Syllabus/Python/Example/loop.py (3.10.0)
File Edit Format Run Options Window Help
print("Iterating over list")
nameList = ["ramesh", "suresh", "rajesh"]
for names in nameList:
    print(names)

print("Iterating over tuple")
nameTuple = ("Rohan", "Sunitha", "sushanth")
for names in nameTuple:
    print(names)

print("Iterating over string")
nameString = "Nilesh"
for names in nameString:
    print(names)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/loop.py =====
Iterating over list
ramesh
suresh
rajesh
Iterating over tuple
Rohan
Sunitha
sushanth
Iterating over string
N
i
l
e
s
h
>>>
```

Loop



Else statement with for loop in python:

- We can also combine else statement with for loop.
- As there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

Syntax:

```
for iterator_variable in sequence:
```

```
    statements
```

```
else:
```

```
    statements
```

Loop



Example:

```
loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)

File Edit Format Run Options Window Help
fruit = "banana"
for x in fruit:
    print(x)
else:
    print("else block is executed")
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: D:\Syllabus\Python\Example\loop.py =====
b
a
n
a
n
a
else block is executed
>>>
```

Loop



Nested for loop:

- Python programming language allows to use one for loop inside another for loop.

Syntax:

for iterator_var in sequence:

for iterator_var in sequence:

statements

statements

Loop



Example:

```
loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)

File Edit Format Run Options Window Help

basket = ["basket1", "basket2", "basket3"]
fruits = ["apple", "orange", "mango"]
print("arranging fruits in each basket")

for x in basket:
    for y in fruits:
        print(x, y)
```

output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help

Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
=====
RESTART: D:\Syllabus\Python\Example\loop.py =====
arranging fruits in each basket
basket1 apple
basket1 orange
basket1 mango
basket2 apple
basket2 orange
basket2 mango
basket3 apple
basket3 orange
basket3 mango
>>> |
```

Loop



While Loop:

- The While loop is used to iterate a part of a program repeatedly until the specified **Boolean Condition is True**. As soon as the Boolean condition turns **False**, The loop **automatically stops**.
- The while loop is considered as a repeating **if statement**.
- If the number of iterations is not fixed, it is **recommended** to use **while loop**.

Syntax:

while condition:

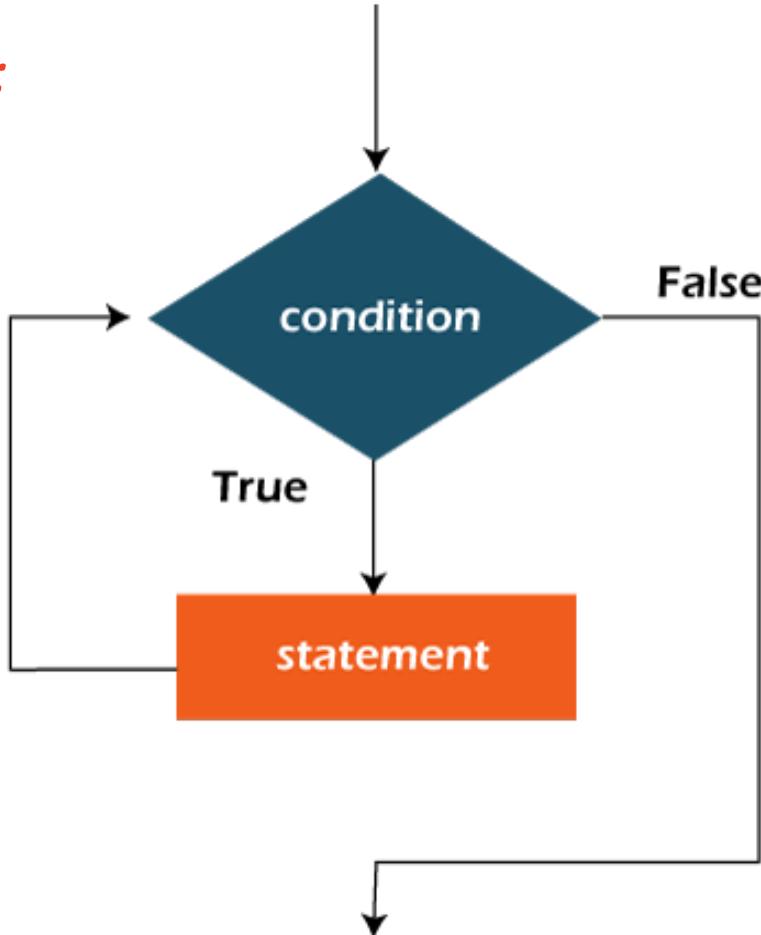
statement

increment/decrement

Loop



Flow chart:



Loop



Example:

```
*loop.py - D:/Syllabus/Python/Example/loop.py (3.10.0)*
File Edit Format Run Options Window Help
sum = 0
number = int(input("Enter the number:"))
while number>0:
    sum += number
    number = int(input("Enter the number"))

print("sum of the given number is:", sum)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/loop.py =====
Enter the number:20
Enter the number:21
Enter the number:12
Enter the number:34
Enter the number:-12
sum of the given number is: 87
>>>
```

Loop



Example:

```
loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)

File Edit Format Run Options Window Help
number = int(input("Enter the number:"))
i = 1
print("Multiplication table of ", i, ": ")
while i<=13:
    print(number, " * ", i, " = ", (number*i))
    i = i + 1
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\Syllabus\Python\Example\loop.py =====
Enter the number:
Multiplication table of  1 :
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
2 * 11 = 22
2 * 12 = 24
2 * 13 = 26
>>>
```

Loop



Else statement in while loop:

- The else clause is only executed when your while condition becomes false.
- If you break out of the loop or if an exception is raised, it won't be executed.

Syntax:

while condition:

statements

else:

statements

Loop



Example:

loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)

File Edit Format Run Options Window Help

```
n = 0
while n<10:
    print("hi loop executed", n , " times")
    n = n+1
else:
    print("else block executed")
```

Output:

IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:\Syllabus\Python\Example\loop.py =====
hi loop executed 0 times
hi loop executed 1 times
hi loop executed 2 times
hi loop executed 3 times
hi loop executed 4 times
hi loop executed 5 times
hi loop executed 6 times
hi loop executed 7 times
hi loop executed 8 times
hi loop executed 9 times
else block executed
>>>
```

Loop



Nested while loop:

- Python programming language allows to use one while loop inside another while loop.

Syntax:

while expression:

while expression:

statement(s)

statement(s)

Loop



Example:

```
*loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)*
File Edit Format Run Options Window Help
i = 2

while i<100:
    j = 2

    while (j <= (i/j)):
        if not(i%j):
            break
        j = j+1
    if(j > i/j):
        print(i, " is prime")
    i = i+1

print("Good Bye! ")
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
=====
RESTART: D:\Syllabus\Python\Example\loop.py =====
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
Good Bye!
```

Loop



Loop control statements:

- Loop control statements change execution from their normal sequence.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Three control statements:

- ✓ Continue statement
- ✓ Break statement
- ✓ Pass statement

Loop



Continue Statement:

- The continue statement in python returns the control to the beginning of the loop.

Example:

```
*loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)*
File Edit Format Run Options Window Help
print("Example for continue statement")

for letter in "python language":
    if letter == 't' or letter == 'g':
        continue
    print("current letter: ", letter)
print("letters g and t are remove from the string ")
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:\Syllabus\Python\Example\loop.py =====
Example for continue statement
current letter: p
current letter: y
current letter: h
current letter: o
current letter: n
current letter:
current letter: l
current letter: a
current letter: n
current letter: u
current letter: a
current letter: e
letters g and t are remove from the string
>>>
```

Loop



break Statement:

- The break statement in python brings the control out of the loop.

Example:

```
loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)
File Edit Format Run Options Window Help
print("Example for continue statement")

for letter in "python language":
    if letter == 't' or letter == 'g':
        break
    print("current letter: ", letter)
print("The loop will be exited when letter t or g is reached")
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:\Syllabus\Python\Example\loop.py =====
Example for continue statement
current letter: p
current letter: y
The loop will be exited when letter t or g is reached
>>> |
```

Loop



pass Statement:

- We use pass statement in python to write empty loops.
- pass is also used for empty control statements, functions and classes.

Example:

```
loop.py - D:\Syllabus\Python\Example\loop.py (3.10.0)
File Edit Format Run Options Window Help
print("Example for continue statement")

for letter in "python language":
    pass
print("current letter: ", letter)
print("Last letter of the value id printed")
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:\Syllabus\Python\Example\loop.py =====
Example for continue statement
current letter: e
Last letter of the value id printed
>>>
```

Thank You



Functions

Function



What is Function?

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

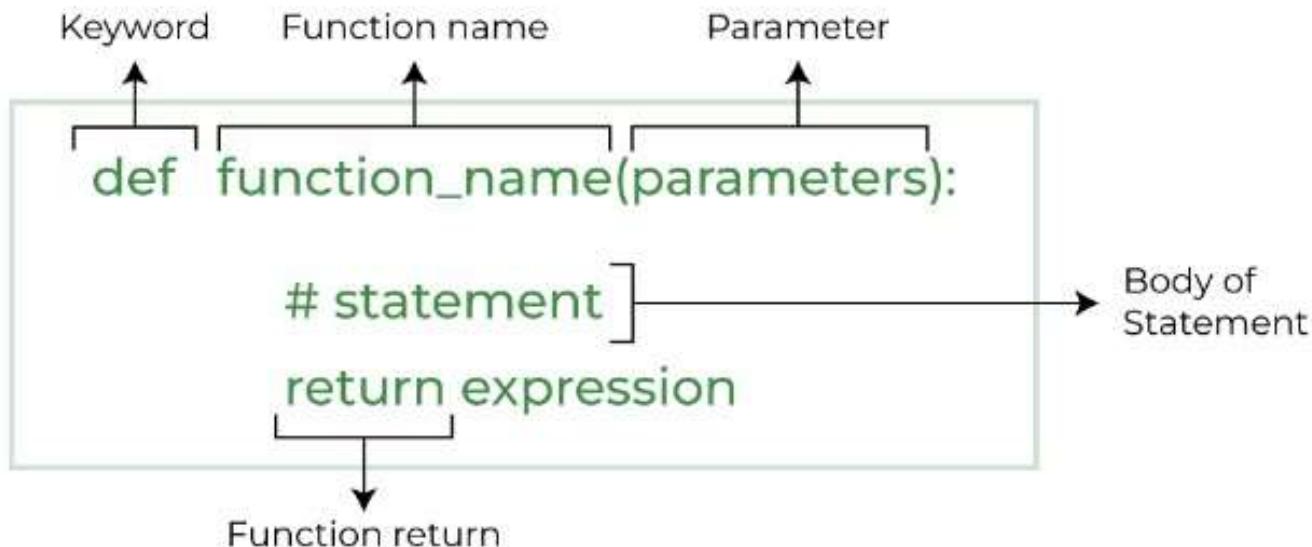
Benefits of using functions:

- Increase code Readability
- Increase code Reusability

Function



Python Function Declaration:



Function



Types of function:

Built – in – function:

- These are standard functions in python that are available to use.
- *For Example:* append(), clear(), add() etc..

User – defined – function:

- We can create our own functions based on our requirements.

Function



Creating a Function:

- We can create a user defined function in python by using the def keyword.
- After creating a function we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

Example:

```
def myfun():
    print("welcome to my function")

myfun()
```

Output:

```
welcome to my function
|
```

Function



Python function with arguments:

- An argument is a value that is accepted by a function.
- If we create a function with arguments, we need to pass the corresponding values while calling them.

Example:

```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print('Sum: ',sum)

add_numbers(5,4)
```

Output:

```
Sum: 9
```

Function



Return statement:

- If we want our function to return some value to a function call we can use return statement.
- The return statement also denotes that the function has ended. Any code after return is not executed.
- A python function may or may not return a value.

Function



Example:

```
# function definition
def find_square(num):
    result = num * num
    return result

# function call
value = int(input("Enter the number to find square value: "))
square = find_square(value)

print('Square:',square)
```

Output:

```
Enter the number to find square value: 2
Square: 4
> |
```

Function



Standard functions:

- Standard library functions are the built-in functions that can be used directly in our program.
- These library functions are defined inside a module and to use them, we must include the module inside our program.

Function



Example:

```
import math

# sqrt computes the square root
value1 = int(input("Enter the number1: "))
value2 = int(input("Enter number2: "))
square_root = math.sqrt(value1)

print("Square Root is",square_root)

# pow() computes the power
power = pow(value1, value2)

print("value1 to the power value2 is",power)
```

Output:

```
Enter the number1: 4
Enter number2: 2
Square Root is 2.0
value1 to the power value2 is 16
> |
```

Thank You



Scopes In Python

Scope



What is a Scope?

- A variable is only available from inside the region it is created. This is known as scope.

Types of scopes:

- Local Scope
- Global Scope
- Naming Variables
- Global keyword

Scope



Local Scope:

- A variable created inside a function belongs to the local scope of that function.
- It can be only used inside that function.

Example:

```
def myfunction():
    x = 200

    print(x)
myfunction()
```

Output:

The screenshot shows the IDLE Shell 3.10.0 interface. The title bar reads "IDLE Shell 3.10.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: D:/Syllabus/Python/Example/scope.py =====
Traceback (most recent call last):
  File "D:/Syllabus/Python/Example/scope.py", line 4, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

Scope



Function Inside Function:

- As explained before the variable inside a function is not available outside the function.
- But it is available for any function inside the function.

Example:

```
def myFunction():
    x = 200
    def myInnerFunction():
        print(x)
    myInnerFunction()
myFunction()
```

Output:

The screenshot shows the IDLE Shell 3.10.0 interface. The title bar reads "IDLE Shell 3.10.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
===== RESTART: D:/Syllabus/Python/Example/scope.py ======
200

Scope



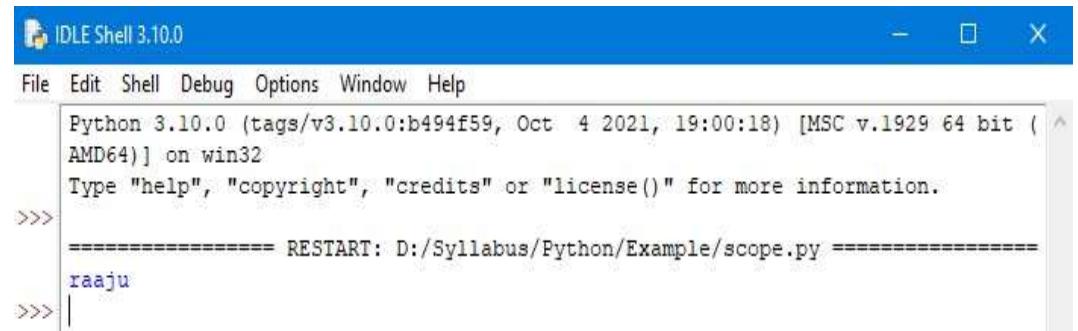
Global Scope:

- A variable created in the main body of the python code is a global variable and belongs to the global scope.
- Global variables are available from within any scope, global and local.

Example:

```
name = "raaju"  
def myFunction():  
    print(name)  
  
myFunction()
```

Output:



The screenshot shows the IDLE Shell interface with the title bar "IDLE Shell 3.10.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The Python version is listed as "Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32". The shell window displays the following text:
>>> Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: D:/Syllabus/Python/Example/scope.py ======
>>> rraaju

Scope



Naming Variables:

- If you operate with the same variable name inside and outside of a function.
- Python will treat them as two separate variables.
- One available in the global scope and one available in the local scope.

Example:

```
x = 300

def myFunction():
    x = 200
    print(x)

myFunction()
print(x)
```

Output:

The screenshot shows the IDLE Shell interface with the following content:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: D:/Syllabus/Python/Example/scope.py =====
200
300
>>> |
```

Scope



Global Keyword:

- If you need to create a global variable, but are stuck in the local scope.
- You can use the global keyword.
- The global keyword makes the variable global.

Example:

```
def myFunction():
    global x
    x = 200

myFunction()
print(x)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: D:/Syllabus/Python/Example/scope.py =====
200
>>>
```

Thank You



Naming Conventions

Naming conventions



Naming conventions:

- Naming Conventions is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, function etc.
- But it is not forced to follow.

Naming Conventions



General rules for python naming conventions:

Rule1: Always give a meaningful full name for any of the python objects.

Don't give names like x, y, or z etc.

Rule2: Don't give space in between object name instead of using underscore(_) when there are more than one-word presents.

Rule3: Better to use camelCase only when it makes sense or else try avoiding more use of camelCase.

Naming conventions



Naming convention for identifiers:

Function:

- We should write the python function name with all lower case characters.
- Do not use uppercase character while naming a function in python.
- Use underscore(_)in between the words instead of space while naming a function.

Example:

```
def example_function():
    print("This is the way a function must be used")
```

Naming conventions



Naming convention for identifiers:

Variable:

- You must start variable name with an alphabet or underscore(_) character.
- A variable name can only contain Alphanumeric (A-Z, 0-9) and underscore(_).
- You cannot start the variable name with a number.
- You cannot use special characters with the variable name such as \$, %, @, #, &, ^ etc.
- Variable names are case sensitive. For example age and Age are two different variables.
- Do not use reserve keywords as a variable name for example class, for, def etc.

Naming conventions



Naming convention for identifiers:

Variable:

Example:

```
#Allowed variable names
```

```
x=2  
y="Hello"  
mypython="PythonGuides"  
my_python="PythonGuides"  
_my_python="PythonGuides"  
_mypython="PythonGuides"  
MYPYTHON="PythonGuides"  
myPython="PythonGuides"  
myPython7="PythonGuides"
```

```
#Variable name not Allowed
```

```
7mypython="PythonGuides"  
-mypython="PythonGuides"  
myPy@thon="PythonGuides"  
my Python="PythonGuides"  
for="PythonGuides"
```

Naming conventions



Naming convention for identifiers:

Class:

- We need to follow the camelCase convention.
- When you are writing any classes for an exception then that name should end with “Error”.
- If you are calling the class from somewhere or callable then, you can give a class name like a function.
- The built-in classes in python are with lowercase.

Naming conventions



Naming convention for identifiers:

Class:

Example:

class MyClass

class Hello

class InputError

class ProperClassNameingConvention

Naming conventions



Naming convention for identifiers:

Package:

- You should use all lowercase while deciding a name for a package.
- If there are multiple words in your package name then they should be separated by an underscore(_).
- It is always better to use a single word for a package name.

Example:

from mypkg import mod1

from example_package import cars

Naming conventions



Naming convention for identifiers:

Global variable:

- You should use all lowercase while deciding a name for a object.
- If there multiple words in your global variable name then they should be separated by an underscore(_).

Example:

```
global my_variable
```

```
global user_name
```

```
global AGE
```

Thank You



Basic Keywords

Basic Keywords



Keywords:

- Keywords are the reserved words in python.
- We cannot use a keyword as a variable name, function name or any other identifier.

Basic Keywords



Value Keywords:

True : The True keyword is used as the Boolean true value in Python code.

False : The false keyword is used as the Boolean false value in Python code.

None : The python None represents no value. In other programming languages, None is represented as null, none, nil.

Basic Keywords



Operator Keywords:

and : The and keyword is used to determine if both the left and right operands are truthy or falsy.

or : The or keyword is used to determine if at least one of the operands is truthy.

not : The not keyword is used to get the opposite Boolean value of a variable.

in : The in keyword will return True or False indicating whether the element was found in the container.

Is : The is keyword determines whether two objects are exactly the same object.

Basic Keywords



Control Flow Keywords:

if : The if statement allows you to write a block of code that gets executed only if the expression after if is truthy.

elif : The elif statement is only valid after an if statement or another elif.

else : The else statement denotes a block of code that should be executed only if the other conditional blocks are false.

Basic Keywords



Iteration Keywords:

for : The for keyword is used to start a for loop. It is used to execute a set of instruction/functions repeatedly when some condition becomes True.

while : The while keyword is used to start a while loop. This loop iterates a part of the program several times.

Basic Keywords



Loop control Keywords:

break : The break keyword is used exit a loop.

continue : The continue keyword is used when you want to skip to the next loop iteration.

else : The else keyword specifies code that should be run if the loop exits normally.

Basic Keywords



Structure Keywords:

def : The def keyword is used to define a function or method of a class.

class: The class keyword is used to define a class.

with : With statement is used to wrap the execution of a block of code within methods defined by the context manager.

as : The as keyword is used to create as alias while importing a module.

Basic Keywords



Returning Keywords:

return : The return keyword is used inside a function to exit it and return a value.

Yield : The yield keyword is used inside a function like a return statement, but yield returns a generator.

Basic Keywords



Import Keywords:

import : The import keyword is used to import modules into the current namespace.

from : the from keyword is used together with import to import something specific from a module.

as : The as keyword is used to alias an imported module or tool.

Thank You



Array

Arrays



What is an array?

- Array is a container which can hold a fix number of items and these items should be of the same type.

Terms used in array:

- ***Element:*** Each item stored in an array is called an element.
- ***Index:*** Each location of an element in an array has a numerical index, which is used to identify the element.

Arrays



Syntax:

```
from array import *
arrayName = array(typecode, [Initializers])
```

- In above example type code are the codes that are used to define the type of value the array will hold.

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
l	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Array



Array Operations:

- Traverse Operation
- Insertion Operation
- Deletion Operation
- Search Operation
- Update Operation

Arrays



Array Operations:

Traverse operation:

- This operation is used to print all the array elements one by one.

Example:

```
from array import *
array1 = array('i', [10, 20, 30, 40, 50])
for i in array1:
    print(i)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, AMD64) on win32
Type "help", "copyright", "credits"
>>> ===== RESTART: D:/Syllab
10
20
30
40
50
>>> |
```

Arrays



Array Operations:

Insertion operation:

- Insertion operation is used to insert one or more data elements into an array.

Example:

```
from array import *

array1 = array('i', [10, 20, 30, 40, 50])

array1.insert(1, 60)

for i in array1:
    print(i)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59
AMD64) ] on win32
Type "help", "copyright", "credits"
>>>
=====
RESTART: D:/Syllabus/arrays.py
10
60
20
30
40
50
>>> |
```

Arrays



Array Operations:

Deletion operation:

- Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Example:

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1.remove(40)

for x in array1:
    print(x)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f5
AMD64) ] on win32
Type "help", "copyright", "credits"
>>> ===== RESTART: D:/Syll
10
20
30
50
>>> |
```

Arrays



Array Operations:

Search operation:

- This operation searches an element using the given index or by the value.

Example:

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
print (array1.index(40))
```

Output:

```
IDLE Shell 3.10.0  
File Edit Shell Debug Options Window Help  
Python 3.10.0 (tags/v3.10.0:b494f5  
AMD64) ] on win32  
Type "help", "copyright", "credits"  
>>>  
===== RESTART: D:/Syll  
3  
>>> |
```

Arrays



Array Operations:

Update operation:

- Update operation refers to updating an existing element from the array.

Example:

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1[2] = 80

for x in array1:
    print(x)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f
AMD64) on win32
Type "help", "copyright", "credit"
>>>
=====
RESTART: D:/Syl
10
20
80
40
50
>>> |
```

Thank You



Type Casting

Type casting



What is Type casting?

- Type casting is a method or process that converts a datatype into another datatype in both ways manually and automatically.
- The automatic conversion is done by the compiler and manual conversion is done by the programmer.

Types of type casting:

- ❖ Implicit Type Casting
- ❖ Explicit Type Casting

Type Casting



Implicit Type Casting:

- This method converts the datatype into another datatype automatically.
- In this process, users don't have to change the types.

Type Casting



Example:

```
typecasting.py - D:/Syllabus/Python/Example/typecasting.py (3.10.0)
File Edit Format Run Options Window Help
# Python program to demonstrate
# implicit type Casting

# Python automatically converts
# a to int
a = 7
print(type(a))

# Python automatically converts
# b to float
b = 3.0
print(type(b))

# Python automatically converts
# c to float as it is a float addition
c = a + b
print(c)
print(type(c))

# Python automatically converts
# d to float as it is a float multiplication
d = a * b
print(d)
print(type(d))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 12 2021, 15:53:35) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or
>>>
=====
RESTART: D:/Syllabus/Py1
<class 'int'>
<class 'float'>
10.0
<class 'float'>
21.0
<class 'float'>
>>> |
```

Type Casting



Explicit Type Casting:

- This method needs user involvement to convert the variable data type into a certain datatype in order to the operation required.
- Type casting can be done with the following datatype functions.
 - *Int():* python int() function take float or string as an argument and returns int type object.
 - *Float():* Python float() function take int or string as an argument and return float type object.
 - *Str():* Python str() function takes float or int as an argument and returns string type object.

Type Casting



Python int to float:

Example:

```
# Python program to demonstrate
# type Casting

# int variable
a = 5

# typecast to float
n = float(a)

print(n)
print(type(n))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f51
AMD64) ] on win32
Type "help", "copyright", "credits"
>>>
=====
RESTART: D:/Syllabus
5.0
<class 'float'>
>>> |
```

Type Casting



Python int to string:

Example:

```
# Python program to demonstrate
# type Casting

# int variable
a = 5

# typecast to str
n = str(a)

print(n)
print(type(n))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f5
AMD64) on win32
Type "help", "copyright", "credits"
=====
RESTART: D:/Syllabu
5
<class 'str'>
|
```

Type Casting



Python float to int:

Example:

```
# Python program to demonstrate
# type Casting

# int variable
a = 5.9

# typecast to int
n = int(a)

print(n)
print(type(n))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f5!
AMD64) ] on win32
Type "help", "copyright", "credits"
=====
RESTART: D:/Syllabus
>>> 5
<class 'int'>
>>> |
```

Type Casting



Python float to string:

Example:

```
# Python program to demonstrate
# type Casting

# int variable
a = 5.9

# typecast to str
n = str(a)

print(n)
print(type(n))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f5
AMD64) ] on win32
Type "help", "copyright", "credits"
=====
RESTART: D:/Syllabus
>>> 5.9
<class 'str'>
>>>
```

Type Casting



Python string to int:

Example:

```
# string variable
a = "5"

# typecast to int
n = int(a)

print(n)
print(type(n))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59,
AMD64) ] on win32
Type "help", "copyright", "credits"
>>> ===== RESTART: D:/Syllabus/
5
<class 'int'>
>>> |
```

Type Casting



Python string to float:

Example:

```
# string variable
a = "5"

# typecast to int
n = float(a)

print(n)
print(type(n))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59,
AMD64) ] on win32
Type "help", "copyright", "credits"
>>> ===== RESTART: D:/Syllabus/
5.0
<class 'float'>
>>> |
```

Thank You



History Of Oops

History of Oops



History of Oops:

- Many people believe that **C++** is the first Oop based language.
- Actually, **SIMULA 1 (1962)** and **SIMULA 2 (1967)** were the Earliest Object Oriented Languages.
- Oops began to flourish after c++ began to entrenched in **1990s**.



Ole – Johan Dahl & Kristen nygaard

Creators of Simula 1 and Simula 2



Bjarne Stroustrup

Creator of C++

Why Oops was Created



Why oops:

- The oop concept was basically designed to overcome the drawback of the previous **Programming Methodologies**, which were not so close to **Real – World Applications**.
- The Demand was **increased**, but still **conventional methods** were used.
- This new approach brought a **revolution** in **Programming Methodology field**.
- **Object – Oriented Programming (Oop)** is nothing but that which allows the writing of programs with **classes** and **real – time objects**.

Advantages of Oops



Advantages of Oops:

- By using **class** we can reuse some facilities rather than building them again and again.
- It is possible to have **multiple Objects**.
- Through **inheritance** we can eliminate **unnecessary codes** and extend the use of **existing classes**.
- The principle of **data hiding** helps for security.
- It is easy to partition the work in a project based.
- Object – oriented systems can be easily upgraded from **small to large** systems.
- **Message passing** techniques for communication between objects.

Bottom-Up Vs Top-Down Model



Sr. No.	Key	Bottom-Up Model	Top-Down Model
1	Focus	In Bottom-Up Model, the focus is on identifying and resolving smallest problems and then integrating them together to solve the bigger problem.	In Top-down Model, the focus is on breaking the bigger problem into smaller one and then repeat the process with each problem.
2	Language	Bottom-Up Model is mainly used by object oriented programming languages like Java, C++ etc.	Top-Down Model is followed by structural programming languages like C, Fortran etc.
3	Redundancy	Bottom-Up model is better suited as it ensures minimum data redundancy and focus is on re-usability.	Top-down model has high ratio of redundancy as the size of project increases.
4	Interaction	Bottom-Up model have high interactivity between various modules.	Top-down model has tight coupling issues and low interactivity between various modules.
5	Approach	Bottom-up model is based on composition approach.	Top-down model is based on decomposition approach.
6	Issues	In Bottom-Up, some time it is difficult to identify overall functionality of system in initial stages.	In Top-Down, it may not be possible to break the problem into set of smaller problems.

Structured programming Vs Oops



Structured programming	Object oriented programming
Code is divided into modules of functions	Code is made up of classes and objects
Less flexibility and abstraction	More flexibility and abstraction
Difficult to modify / manage	Easy to modify / manage
Main function calls other functions	Objects communicate by passing messages
Data is not secured	Data is secured
Less reusability of code	More reusability of code
Top – Down approach	Bottom – up approach

Terminologies of Oops



Terminologies of Oops:

- Classes
- Object
- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

Terminologies of Oops



Classes:

A class is an template used to create objects and to define object data types and methods.

Objects:

An object is an instance of an class. Each object has State (data) and behavior (code).

Encapsulation:

Encapsulation refers to integrating data (variables) and code(methods) into a single unit.

Terminologies of Oops



Data Abstraction:

Data abstraction is a programming and design tool that displays basic information about a device while hiding its internal functions.

Inheritance:

Inheritance is a mechanism in which one class acquires the property of another class.

Polymorphism:

Polymorphism is the ability of any data to be processed in more than one form.

Thank You



Class and Object

Class and Object



What is a Class?

- A class is a user – defined *blueprint or prototype* from which objects are created.
- Classes separates functions and fields based on their categories.
- Suppose a class is a prototype of a building.
- A building contains all the details about the floor, rooms, door, 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 many objects of this class

Class and Object



Syntax:

```
class ClassName:
```

```
    # class definition
```

Example:

```
class bike:
```

```
    model = "R3"
```

```
    brand = "Yamaha"
```

Class and Object



What is Object?

- An Object is an instance of a Class.
- A class is like a blueprint while an instance is a copy of the class with actual values.

An object consists of:

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

Class and Object

Declaring Class Objects:

- When an object of a class is created, the class is said to be instantiated.
- All the instances share the attributes and the behavior of the class.
- But the values of those attributes are unique for each object.
- A single class may have any number of instances.



Class and Object



Example:

Code:

```
# demonstrate instantiating
# a class
class Dog:

    # A simple class
    # attribute
    attr1 = "mammal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)

# Driver code
# Object instantiation
pug = Dog()

# Accessing class attributes
# and method through objects
print(pug.attr1)
pug.fun()
```

Output:

IDLE Shell 3.10.0

File Edit Shell Debug Options Window Help

```
Python 3.10.0 (tags/v3.10.0:b494f
AMD64) ] on win32
Type "help", "copyright", "credit",
>>>
=====
RESTART: D:/Syllab
mammal
I'm a mammal
I'm a dog
>>> |
```

Class and Object



Self parameter:

- SELF represents the instance of class.
- This handy keyword allows you to access variables, attributes and methods of a defined class in python.
- The self parameter doesn't have to be named "self", as we can call it by any other name.

Class and Object



Example:

Code:

```
class Cat:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def info(self):  
        print(f"I am a cat. My name is", self.name, ". I am", self.age, " years old.")  
  
    def make_sound(self):  
        print("Meow")  
  
cat1 = Cat('Andy', 2)  
cat2 = Cat('Phoebe', 3)  
cat1.info()  
cat2.info()
```

Output:

The screenshot shows the IDLE Shell interface with the title bar "IDLE Shell 3.10.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays Python code and its execution output. The code defines a class 'Cat' with methods 'info' and 'make_sound'. It then creates two instances, 'cat1' and 'cat2', and calls their 'info' methods. The output shows the class's behavior: it prints a message with the cat's name and age, and also prints a "Meow" sound.

```
IDLE Shell 3.10.0  
File Edit Shell Debug Options Window Help  
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:41)  
[PyPy 6.3.1 (6.3.1+tags/v6.3.1-10-gd3f9c1b, Oct 4 2021, 19:00:41)  
AMD64] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> ===== RESTART: D:/Syllabus/Python/Example/class.py =====  
I am a cat. My name is Andy . I am 2 years old.  
I am a cat. My name is Phoebe . I am 3 years old.  
>>>
```

Class and Object



Accessing class from another file:

Step1: create a file named first with class name as First and enter the following code in it.

```
first.py
1 class First:
2     def name(self):
3         print("Hi my name is first....")
4
```

Step2: then create a file named second in the same folder where first is located.

first	8/26/2023 11:38 AM	Python File	1 KB
second	8/26/2023 11:41 AM	Python File	0 KB

Class and Object



Accessing class from another file:

Step3: Write the following code in the second file to create object for class First.

```
from first import First

#creating object for class First
fir = First()

#calling method name from First
fir.name()
```

Step4: Now, run the second file.

```
Hi my name is first....
```

Thank You



Constructor



Constructor:

What is Constructor?

- Constructors are generally used for instantiating an object.
- The task of constructors is to initialize(assign value) to the data members of the class when an object of the class is created.
- In python the `__init__() method` is called the constructor and is always called when an object is created.

Types of Constructors:

- Non-parameterized Constructor
- Parameterized Constructor

Constructor



__init__() method:

- The *__init__* method is similar to constructor in c++ and Java.
- Constructors are used to initializing the object's state.
- Like methods, a constructor also contains a collection of statements that are executed at the time of Object creation.
- It runs as soon as an object of a class is instantiated.

Syntax:

```
def __init__(self):
```

```
#body of the constructor
```

Constructor



Example:

```
# Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Nikhil')
p.say_hi()
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59
AMD64) ] on win32
Type "help", "copyright", "credits"
>>> ===== RESTART: D:/Syllabus.
Hello, my name is Nikhil
>>> |
```

Constructor



Non parameterized Constructor:

- The non-parameterized constructor is a simple constructor which doesn't accept any arguments.
- Its definition has only one argument which is a reference to the instance being constructed.

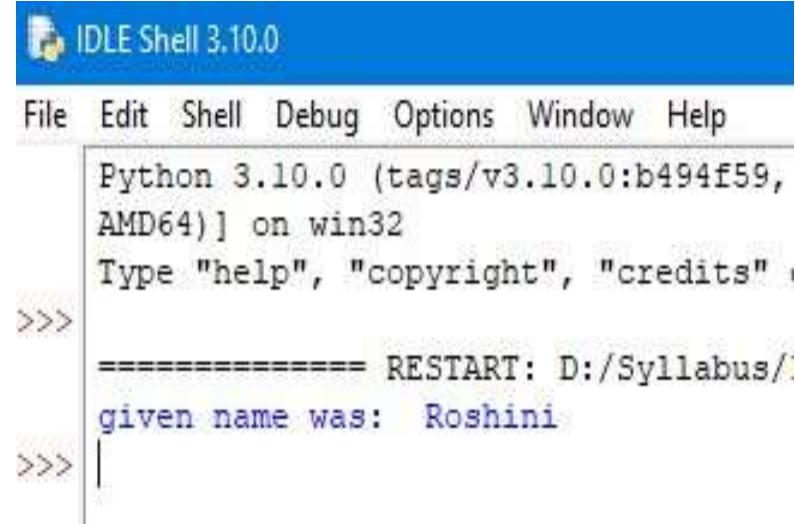
Constructor



Example:

```
class constructor:  
    # default constructor  
    def __init__(self):  
        self.name = "Roshini"  
  
    # a method for printing data members  
    def print_Name(self):  
        print("given name was: ", self.name)  
  
#creating object of the class  
obj1 = constructor()  
  
#calling the instance method using the object obj1  
obj1.print_Name()
```

Output:



```
IDLE Shell 3.10.0  
File Edit Shell Debug Options Window Help  
Python 3.10.0 (tags/v3.10.0:b494f59, AMD64) ] on win32  
Type "help", "copyright", "credits" ,  
>>>  
===== RESTART: D:/Syllabus/  
given name was: Roshini  
>>>
```

Constructor



Parameterized Constructor:

- Constructor with parameters is known as parameterized constructor.
- The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Constructor



Example:

```
class Addition:  
    first = 0  
    second = 0  
    answer = 0  
  
    # parameterized constructor  
    def __init__(self, f, s):  
        self.first = f  
        self.second = s  
  
    def display(self):  
        print("First number = " + str(self.first))  
        print("Second number = " + str(self.second))  
        print("Addition of two numbers = " + str(self.answer))  
  
    def calculate(self):  
        self.answer = self.first + self.second  
  
# creating object of the class  
# this will invoke parameterized constructor  
obj1 = Addition(1000, 2000)  
  
# creating second object of same class  
obj2 = Addition(10, 20)  
  
# perform Addition on obj1  
obj1.calculate()  
  
# perform Addition on obj2  
obj2.calculate()  
  
# display result of obj1  
obj1.display()  
  
# display result of obj2  
obj2.display()
```

Output:

```
IDLE Shell 3.10.0  
File Edit Shell Debug Options Window Help  
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  
AMD64) ] on win32  
Type "help", "copyright", "credits" or "  
>>>  
===== RESTART: D:/Syllabus/Pyth  
First number = 1000  
Second number = 2000  
Addition of two numbers = 3000  
First number = 10  
Second number = 20  
Addition of two numbers = 30  
>>> |
```

Thank You



Packages

Packages



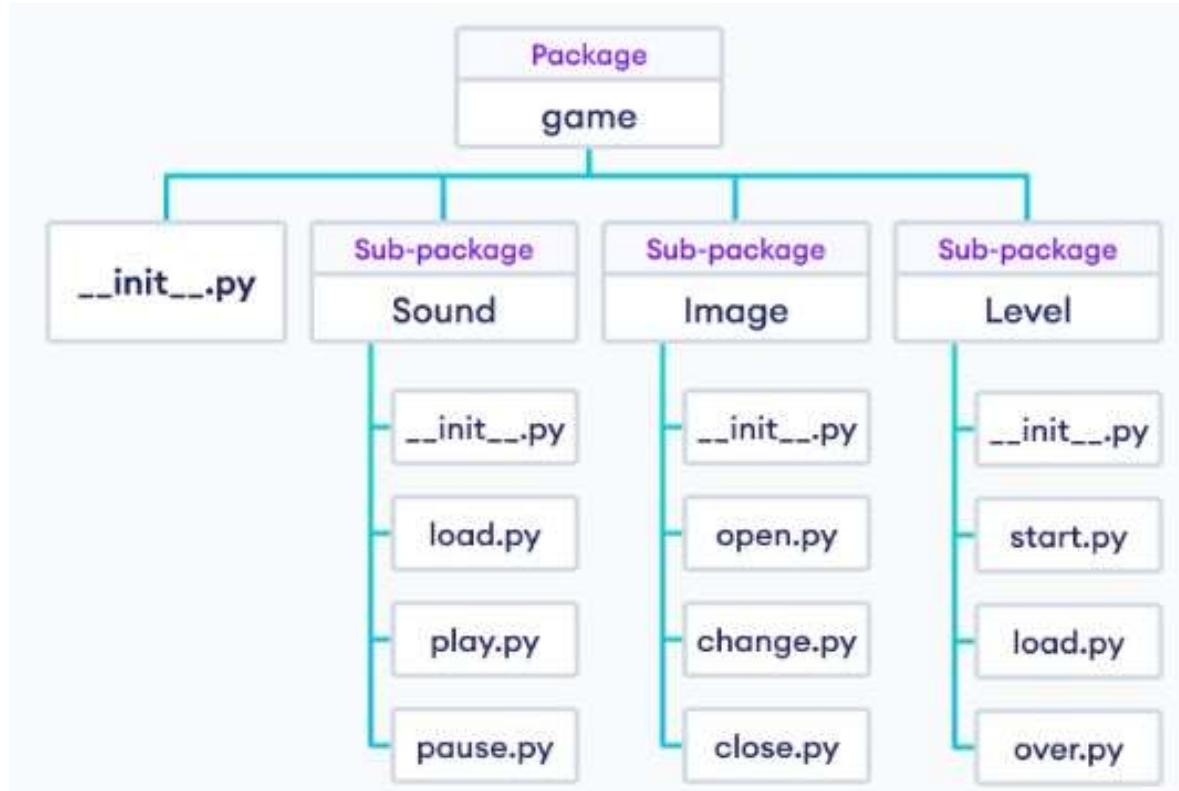
What is a Package?

- A package is a container that contains various functions to perform specific tasks.
- While working on big projects, we have to deal with a large amount of code and writing everything together in the same file will make our code look messy.
- Instead we can separate our code into multiple files by keeping the related code together in packages.
- We can use package whenever we need it in our projects. This way we can also reuse our code.

Packages



Package Model Structure:



Packages



Creating package:

- Create a folder named mypackage.
- Inside the folder create an empty Python file named `__init__.py`.
- Then create two files named mod1 and mod2.

This PC > New Volume (D:) > Syllabus > Python > Example

Name	Date modified	Type	Size
Example	5/22/2023 3:47 PM	File folder	
mypackage	5/22/2023 3:47 PM	File folder	

PC > New Volume (D:) > Syllabus > Python > Example > mypackage

Name	Date modified	Type	Size
<code>__init__</code>	5/22/2023 4:09 PM	Python File	0 KB
<code>mod1</code>	5/22/2023 4:09 PM	Python File	0 KB
<code>mod2</code>	5/22/2023 4:10 PM	Python File	0 KB

Packages



- Write the following code in *mod1.py*.

```
*mod1.py - D:\Syllabus\Python\Example\mypackag
File Edit Format Run Options Window Help
def pkg():
    print("Welcome to packages")
```

- Write the following code in *mod2.py*.

```
mod2.py - D:\Syllabus\Pyth
File Edit Format Run Options Help
def sum(a, b):
    return a+b
```

- Write the following code in *__init__.py*.

```
*__init__.py - D:\Syllabus\Python\Example\__init__.py (3.10.0)*
File Edit Format Run Options Window Help
from mypackage import mod1
from mypackage import mod2

mod1.pkg()
res = mod2.sum(1, 2)
print(res)
```

Packages



- Now, Run the `__init__.py` file.

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494
AMD64) ] on win32
Type "help", "copyright", "credi
>>>
=====
RESTART: D:\Syl
Welcome to packages
3
>>>
```

- We have executed `__init__.py` only and `mod1.py`, `mod2.py` has been executed.

Packages



Understanding __init__.py:

- __init__.py helps the python interpreter recognize the folder as a package.
- It also specifies the resource to be imported from the modules.
- If the __init__.py is empty this means that all the functions of the modules will be imported.
- We can also specify the functions from each module to be made available.

Packages



We can also import the specific function from the module:

Example:

```
*__init__.py - D:\Syllabus\Python\Example\__init__.p
File Edit Format Run Options Window Help
from mypackage.mod1 import pkg
from mypackage.mod2 import sum

pkg()
res = sum(4, 7)
print(res)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window
Python 3.10.0 (tags/v3.10.0:k
AMD64) ] on win32
Type "help", "copyright", "ci
>>>
=====
RESTART: D:\_
Welcome to packages
11
>>> |
```

Packages



Advantages:

- Packages are useful to arrange all the classes & interface performing the same task.
- Packages hide the classes & interfaces in a separate subdirectory, so that accidental deletion of classes & interfaces will not take place.
- The classes & interfaces of a packages are isolated from the classes & interfaces of another packages.
- Reusability nature of packages makes programming easy.

Thank You



Access Modifier

Access Modifier



Access Modifiers:

- Access Modifiers are used to restrict the access of the class member variable and methods from outside the class.
- Python uses '_' symbol to determine the access control for a specific data members or a member function of a class.
- Access modifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.

Types of Access Modifiers:

- Public Access Modifiers
- Protected Access Modifiers
- Private Access Modifiers

Access Modifier



Public Access Modifier:

- By default the member variables and methods are public which means they can be accessed from anywhere outside or inside the class.
- No public keyword is required to make the class or methods and properties public.

Access Modifier



Example:

```
class Student:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def display(self):  
        print("Name:", self.name)  
        print("Age:", self.age)  
  
s = Student("John", 20)  
s.display()
```

Output:

```
IDLE Shell 3.10.0  
File Edit Shell Debug Options  
Python 3.10.0 (tags/v3.10-0-  
AMD64) ] on win32  
Type "help", "copyright"  
  
===== RESTART: D:  
Name: John  
Age: 20
```

Access Modifier



Private Access Modifier:

- Class properties and methods with private access modifier can only be accessed within the class where they are defined and cannot be accessed outside the class.
- In python private properties and methods are declared by adding a prefix with two underscores('__')before their declaration.

Access Modifier



Example:

```
class BankAccount:  
    def __init__(self, account_number, balance):  
        self.__account_number = account_number  
        self.__balance = balance  
  
    def __display_balance(self):  
        print("Balance:", self.__balance)  
  
b = BankAccount(1234567890, 5000)  
b.__display_balance()
```

Output:

```
IDLE Shell 3.10.0  
File Edit Shell Debug Options Window Help  
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: D:\Syllabus\Python\Example\Example\accessModifier.py =====  
Traceback (most recent call last):  
  File "D:\Syllabus\Python\Example\Example\accessModifier.py", line 11, in <module>  
    b.__display_balance()  
AttributeError: 'BankAccount' object has no attribute '__display_balance'  
>>>
```

Access Modifier



Protected Access Modifier:

- Class properties and methods with protected access modifier can be accessed within the class and from the class that inherits the protected class.
- In python, protected members and methods are declared using single underscore('_') as prefix before their names.

Access Modifiers



Example:

```
class Person:  
    def __init__(self, name, age):  
        self._name = name  
        self._age = age  
  
    def _display(self):  
        print("Name:", self._name)  
        print("Age:", self._age)  
  
class Student(Person):  
    def __init__(self, name, age, roll_number):  
        super().__init__(name, age)  
        self._roll_number = roll_number  
  
    def display(self):  
        self._display()  
        print("Roll Number:", self._roll_number)  
  
s = Student("John", 20, 123)  
s.display()
```

Output:

```
IDLE Shell 3.10.0  
File Edit Shell Debug Options  
Python 3.10.0 (tags/v  
AMD64)] on win32  
Type "help", "copyrig  
>>>  
===== RESTART: D:  
Name: John  
Age: 20  
Roll Number: 123  
>>> |
```

Thank You



Encapsulation

Encapsulation



What is Encapsulation?

- Encapsulation is one of the cornerstone concepts of OOP.
- Encapsulation is used to wrap up both data and methods into one single unit.
- The way that data and methods are organized does not matter to the end – user.
- The user is only concerned about the right way to provide input and expects a correct output on the basis of the input provided.



Encapsulation



Importance of Encapsulation:

- To hide the internal implementation details of the class.
- Can safely modify the implementation without worrying breaking the existing code that uses the class.
- Protect class against accidental/ willful stupidity.
- Easier to use and understand.

Encapsulation



Encapsulation using public members:

- As the name suggests, the public modifier allows variables and functions to be accessible from anywhere within the class and from any part of the program.
- All member variables have the access modifier as public by default.

Encapsulation



Example:

```
# illustrating public members & public access modifier
class pub_mod:
    # constructor
    def __init__(self, name, age):
        self.name = name;
        self.age = age;

    def Age(self):
        # accessing public data member
        print("Age: ", self.age)

# creating object
obj = pub_mod("Jason", 35);
# accessing public data member
print("Name: ", obj.name)
# calling public member function of the class
obj.Age()
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options
Python 3.10.0 (tags/v3
AMD64) ] on win32
Type "help", "copyright"
>>>
=====
RESTART: D:/
Name: Jason
Age: 35
>>> |
```

Encapsulation



Encapsulation using protected members:

- What sets protected members apart from private members is that they allow the members to be accessed within the class and allow them to be accessed by the sub – classes involved.
- In python, we demonstrate a protected member by prefixing with an underscore ('_') before its name.
- If the members have a protected access specifier it can also be referenced then within the class and the subsequent sub-clas.

Encapsulation



Example:

```
# illustrating protected members & protected access modifier
class details:
    _name="Jason"
    _age=35
    _job="Developer"
class pro_mod(details):
    def __init__(self):
        print(self._name)
        print(self._age)
        print(self._job)

# creating object of the class
obj = pro_mod()
# direct access of protected member
print("Name:",obj.name)
print("Age:",obj.age)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Syllabus/Python/Example/Example/encapsulation.py =====
Jason
35
Developer
Traceback (most recent call last):
  File "D:/Syllabus/Python/Example/Example/encapsulation.py", line 15, in <module>
    print("Name:",obj.name)
AttributeError: 'pro_mod' object has no attribute 'name'. Did you mean: '_name'?
>>>
```

Encapsulation



Encapsulation using private members:

- The private access modifier allows member methods and variables to be accessed only within the class.
- To specify a private access modifier for a member, we make use of the double underscore('__').

Encapsulation



Example:

```
# illustrating private members & private access modifier
class Rectangle:
    __length = 0 #private variable
    __breadth = 0#private variable
    def __init__(self):
        #constructor
        self.__length = 5
        self.__breadth = 3
    #printing values of the private variable within the class
    print(self.__length)
    print(self.__breadth)

rect = Rectangle() #object created
#printing values of the private variable outside the class
print(rect.length)
print(rect.breadth)
```

Output:

```
DLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Syllabus/Python/Example/Example/encapsulation.py =====
5
3
Traceback (most recent call last):
  File "D:/Syllabus/Python/Example/Example/encapsulation.py", line 15, in <module>
    print(rect.length)
AttributeError: 'Rectangle' object has no attribute 'length'
>>>
```

Encapsulation



Example:

```
class add:  
    def __add(self,f,s):  
        print(f+s)  
    def display(self,f,s):  
        self.__add(f,s)  
        print("display called")
```

Output:

```
5  
display called
```

```
from encap.addtion import add  
  
obj = add()  
obj.display(2,3)
```

Encapsulation



Advantages of Encapsulation:

- Code Reusability
- Data hiding
- Improved Maintainability
- Easier to understand
- Better control over class properties
- Better class design

Thank You



Polymorphism

Polymorphism



What is polymorphism?

- Polymorphism means “*Many Forms*”.
- That is, The same entity can perform different operations in different scenarios.
- It occurs when we have classes that are related to each other by inheritance.

Polymorphism



Method Overloading :

- Method Overloading is the class having methods that have the same name with different arguments.
- Arguments will be differed from number of arguments and type of argument.
- It is used in a single class.

Polymorphism



Example:

```
from multipledispatch import dispatch

@dispatch(int,int)
def add(fn,sn):
    return fn+sn

@dispatch(int,int,int)
def add(fn,sn,tn):
    return fn+sn+tn

@dispatch(int,int,int,int)
def add(fn,sn,tn,ffn):
    return fn+sn+tn+ffn

result = add(3,4,5)
print(result)
```

Output:



12

Polymorphism



Method Overriding :

- Method Overriding in python is when you have two methods with the same name that each perform different tasks.
- This is an important feature of inheritance in python.
- In method overriding the child class can change its functions that are defined by its ancestral classes.

Polymorphism



Example:

```
class Parent1:  
    def display(self):  
        print("Inside Parent1")
```

```
class Parent2:  
    def display():  
        print("Inside Parent2")
```

```
from parent import Parent1  
from parent2 import Parent2  
  
class Child(Parent1,Parent2):  
    def show(self):  
        print("inside child")  
  
obj = Child()  
obj.show()  
obj.display()
```

Output:

```
inside child  
Inside Parent1
```

Thank You



Inheritance

Inheritance



What is Inheritance?

- Inheritance is the ability to ‘inherit’ features or attributes from already written classes into newer classes we make.
- These features and attributes are defined data structures and the functions we can perform with them.
- The new class copies all the older class’s functions and attributes without rewriting the syntax in the new classes.
- The old class is called as base class.
- The new class is called as derived classes.

Syntax:

```
class DerivedClass(BaseClass):  
    # class definition
```

Inheritance



Types of Inheritance:

There are five types of inheritance, They are:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

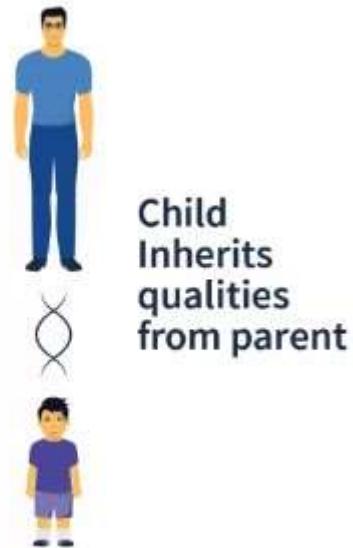
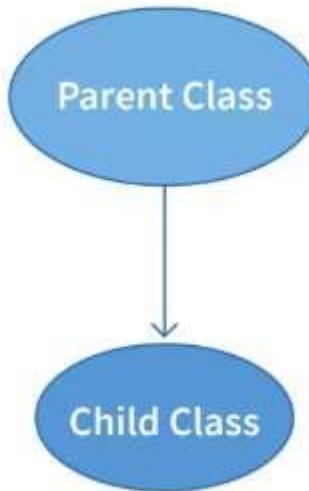


Inheritance



Single Inheritance:

- Single inheritance is the simplest form of inheritance where a single child class is derived from a single parent class.
- It is also known as *Simple Inheritance*.



Inheritance



Example:

```
# single inheritance example

class parent:                  # parent class
    def func1(self):
        print("Hello Parent")

class child(parent):
    # child class
    def func2(self):          # we include the parent class
        # as an argument in the child
        # class

# Driver Code
test = child()                 # object created
test.func1()                    # parent method called via child object
test.func2()                    # child method called
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window
Python 3.10.0 (tags/v3.10-0+g8e8c931) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright" or "credits" for more information.
>>>
=====
RESTART: D:/S
Hello Parent
Hello Child
>>> |
```

Single Inheritance



Inheriting from new file:

- Create a file named parent inside a folder named parent and write the following codes in it.

```
class Parent:  
    def habbits(self):  
        print("I am parent i have my own habbit")
```

- Now create a file named child outside the parent folder and write the following codes.

```
from parent.parent import Parent  
  
class Child(Parent):  
    def character(self):  
        print("I have my own character as well as my fathers character")  
  
obj = Child()  
obj.habbits()  
obj.character()
```

Inheritance



Inheriting from new file:

- Run the child file.

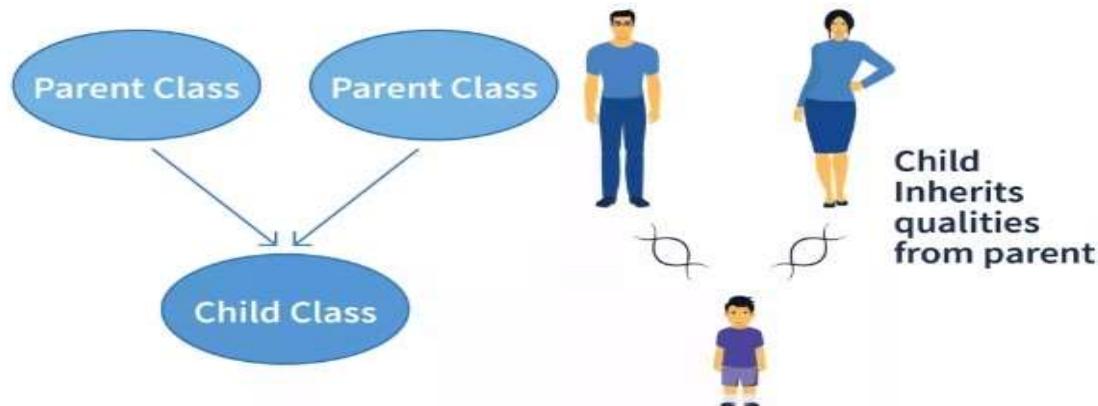
```
I am parent i have my own habbit  
I have my own character as well as my fathers character
```

Inheritance



Multiple Inheritance:

- In multiple Inheritance, a single child class is inherited from two or more parent classes.
- It means the child class has access to all the parent classes methods and attributes.
- If two parents have the same “named” methods, the child class performs the method of the first parent in order of reference.



Inheritance



Example:

```
# multiple inheritance example

class parent1:                      # first parent class
    def func1(self):
        print("Hello Parent1")

class parent2:                      # second parent class
    def func2(self):
        print("Hello Parent2")

class parent3:                      # third parent class
    def func2(self):                 # the function name is same as parent2
        print("Hello Parent3")

class child(parent1, parent2, parent3): # child class
    def func3(self):                 # we include the parent classes
        print("Hello Child")         # as an argument comma separated

# Driver Code
test = child()                      # object created
test.func1()                         # parent1 method called via child
test.func2()                         # parent2 method called via child instead of parent3
test.func3()                         # child method called
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Windows
Python 3.10.0 (tags/v3.10.0b2-7521.d36cfcc-AMD64) [on win32]
Type "help", "copyright", 'credits' or 'license' for more information
>>>
=====
RESTART: D:/Syl...
Hello Parent1
Hello Parent2
Hello Child
>>> |
```

Inheritance



Example:

Parent1.py:

```
class Parent1:  
    def display1(self):  
        print("Hi i am parent1")
```

Parent2.py:

```
class Parent2:  
    def display2(self):  
        print("Hi i am parent2")
```

Inheritance



Example:

Parent3.py:

```
class Parent3:
    def display2(self):
        print("Hi i am parent3")
```

Output:

```
Hi i am parent1
Hi i am parent2
Hi i am parent2
```

child.py:

```
from parent.parent1 import Parent1
from parent.parent2 import Parent2
from parent.parent3 import Parent3
class Child(Parent1,Parent2,Parent3):
    def character(self):
        print("I have my own character as well as my fathers character")

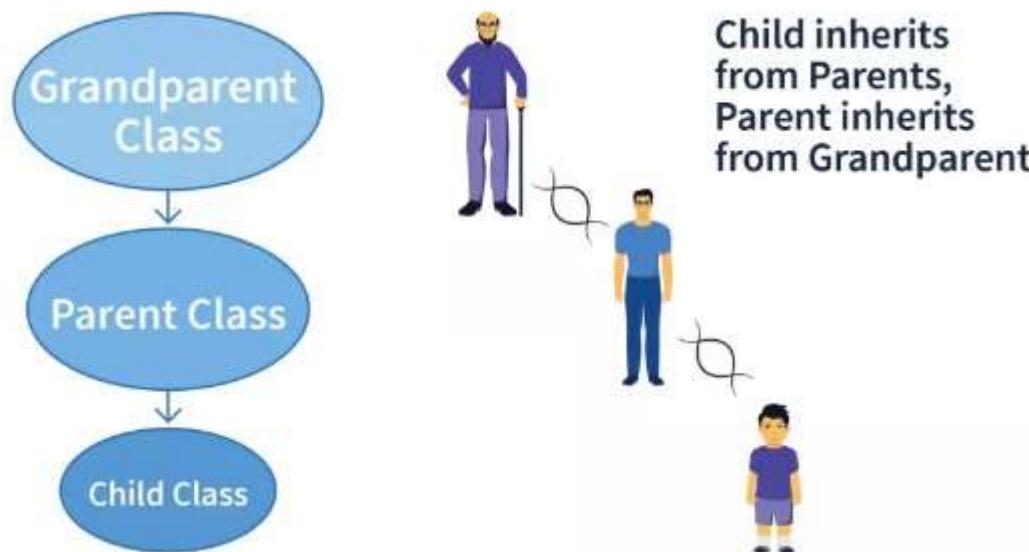
obj = Child()
obj.display1()
obj.display2()
obj.display2()
```

Inheritance



Multilevel Inheritance:

- In multilevel inheritance, we go beyond just a parent – child relation.
- Here we can have multiple levels where the parent class itself is derived from another class.



Inheritance



Example:

```
# Multilevel Inheritance
class grandparent:           # first level
    def func1(self):
        print("Hello Grandparent")

class parent(grandparent):    # second level
    def func2(self):
        print("Hello Parent")

class child(parent):          # third level
    def func3(self):
        print("Hello Child")

# Driver Code
test = child()               # object created
test.func1()                  # 3rd level calls 1st level
test.func2()                  # 3rd level calls 2nd level
test.func3()                  # 3rd level calls 3rd level
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Wind
Python 3.10.0 (tags/v3.10.
AMD64) ] on win32
Type "help", "copyright",
>>>
=====
RESTART: D:/Syl
Hello Grandparent
Hello Parent
Hello Child
>>> |
```

Inheritance



Example:

Grandparent.py:

```
class grandparent:  
    def display1(self):  
        print("Hi i am Grandparent i have my own behaviours")
```

Parent.py:

```
from parent.grandparent import grandparent  
class Parent(grandparent):  
  
    def display2(self):  
        print("Hi i am parent i have my own behaviours aswell as my paents")
```

Inheritance



Example:

child.py:

```
from parent.parent import Parent
class Child(Parent):
    def character(self):
        print("I have my own character as well as my father and grandfather's character")

obj = Child()
obj.display1()
obj.display2()
obj.character()
```

Output:

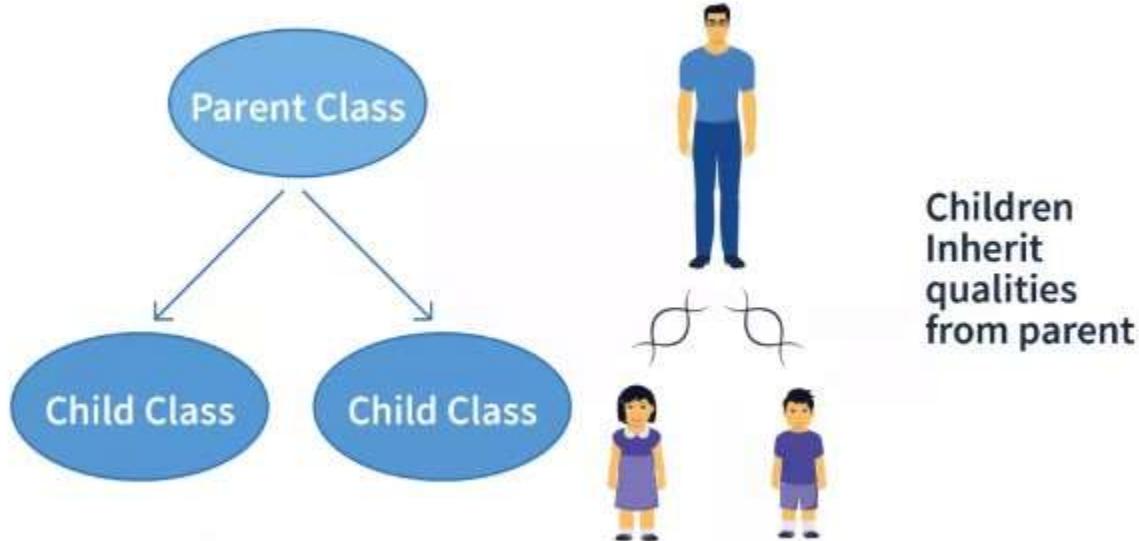
```
Hi i am Grandparent i have my own behaviours
Hi i am parent i have my own behaviours aswell as my paents
I have my own character as well as my father and grandfather's character
```

Inheritance



Hierarchical Inheritance:

- Hierarchical Inheritance is the right opposite of multiple Inheritance.
- It means that there are multiple derived child classes from a single parent class.



Inheritance



Example:

```
# hierarchical inheritance example

class parent:                      # parent class
    def func1(self):
        print("Hello Parent")

class child1(parent):               # first child class
    def func2(self):
        print("Hello Child1")

class child2(parent):               # second child class
    def func3(self):
        print("Hello Child2")

# Driver Code
test1 = child1()                  # objects created
test2 = child2()

test1.func1()                      # child1 calling parent method
test1.func2()                      # child1 calling its own method

test2.func1()                      # child2 calling parent method
test2.func3()                      # child2 calling its own method
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Wind
Python 3.10.0 (tags/v3.10.
AMD64) ] on win32
Type "help", "copyright",
>>>
=====
RESTART: D:/Syl
Hello Parent
Hello Child1
Hello Parent
Hello Child2
>>> |
```

Inheritance



Example:

Parent.py:

```
class Parent:  
    def display2(self):  
        print("Hi i am parent i have my own behaviours aswell as my paents")
```

Child1.py:

```
from parent.parent import Parent  
class child1(Parent):  
    def display3(self):  
        print("Im child1 and i inherit my parent ")  
  
obj = child1()  
obj.display3()  
obj.display2()
```

Output:

```
Im child1 and i inherit my parent  
Hi i am parent i have my own behaviours aswell as my paents
```

Inheritance



Example:

Parent.py:

```
class Parent:  
    def display2(self):  
        print("Hi i am parent i have my own behaviours aswell as my paents")
```

Child2.py:

```
from parent.parent import Parent  
class Child2(Parent):  
    def character(self):  
        print("I am child 2 and i inherit my parent")  
  
obj = Child2()  
obj.character()  
obj.display2()
```

Output:

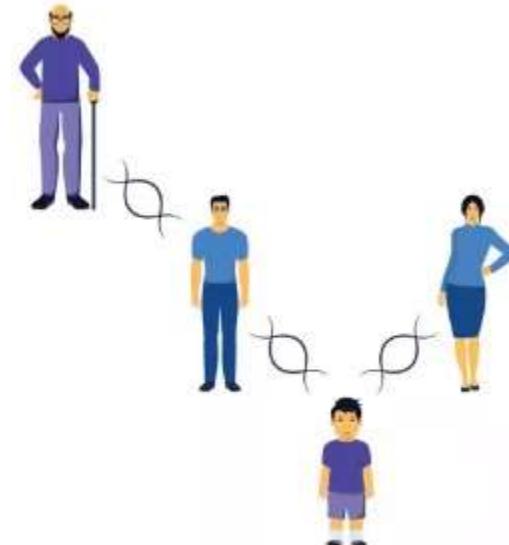
```
I am child 2 and i inherit my parent  
Hi i am parent i have my own behaviours aswell as my paents
```

Inheritance



Hybrid Inheritance:

- Hybrid Inheritance is the mixture of two or more different types of inheritance.
- Here we can have many relationships between parent and child classes with multiple levels.



**Child inherits
from grandparents
and parents**

Inheritance



Example:

```
# hybrid inheritance example

class parent1:                                # first parent class
    def func1(self):
        print("Hello Parent")

class parent2:                                # second parent class
    def func2(self):
        print("Hello Parent")

class child1(parent1):                         # first child class
    def func3(self):
        print("Hello Child1")

class child2(child1, parent2):                  # second child class
    def func4(self):
        print("Hello Child2")

# Driver Code
test1 = child1()                               # object created
test2 = child2()

test1.func1()                                  # child1 calling parent1 method
test1.func3()                                  # child1 calling its own method

test2.func1()                                  # child2 calling parent1 method
test2.func2()                                  # child2 calling parent2 method
test2.func3()                                  # child2 calling child1 method
test2.func4()                                  # child2 calling its own method
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options
Python 3.10.0 (tags/v3.10.0~alpha1-1-ge0d9675) [MSC v.1932 64 bit (AMD64)] on win32
Type "help", "copyright" or "credits" for more information.

>>> ===== RESTART: D:\PycharmProjects\Python\HybridInheritance.py =====
Hello Parent
Hello Child1
Hello Parent
Hello Parent
Hello Child1
Hello Child2
```

Inheritance



Example:

Grandparent.py:

```
class grandparent:  
    def display1(self):  
        print("Hi i am Grandparent i have my own behaviours")
```

Parent.py:

```
from parent.grandparent import grandparent  
  
class Parent(grandparent):  
  
    def display2(self):  
        print("Hi i am parent i have my own behaviours aswell as my paents")
```

Inheritance



Example:

child1.py:

```
from parent.parent import Parent

class child1(Parent):
    def display3(self):
        print("Im child1 and i inherit my parent ")
```

Parent3.py:

```
class Parent3:

    def display4(self):
        print("Hi i am parent3")
```

Inheritance



Example:

child2.py:

```
from parent.parent3 import Parent3
from parent.child1 import child1
class Child2(Parent3, child1):
    def character(self):
        print("I am child 2 and i inherit my parent")

obj = Child2()
#child2
obj.character()
#grandparent
obj.display1()
#parent
obj.display2()
#child1
obj.display3()
#parent3
obj.display4()
```

Output:

```
I am child 2 and i inherit my parent
Hi i am Grandparent i have my own behaviours
Hi i am parent i have my own behaviours aswell as my paents
Im child1 and i inherit my parent
Hi i am parent3
```

Inheritance



Special Functions in Inheritance:

Super() Function:

- Method overriding is an ability of any object – oriented programming language that allows a subclass or child class to provide a specific implementation of a method already provided by one of its superclasses or parent classes.
- In python method overriding is done by using super() function.

Inheritance



Example:

```
class first:  
    def display(self):  
        print("HI i am first classs")
```

Output:

```
HI i am first classs  
Hi i am second class
```

```
from overloading.parent import first  
  
class second(first):  
    def display(self):  
        super().display()  
        print("Hi i am second class")  
  
obj=second()  
obj.display()
```

Inheritance



Issubclass():

- The issubclass() function is a convenient way to check whether a class is the child of the parent class.
- It checks if the first class is derived from the second class.

Example:

```
# issubclass() example

class parent:          # parent class
    def func1():
        print("Hello Parent")

class child(parent):   # child class
    def func2():
        print("Hello Child")

# Driver Code

print(issubclass(child,parent))      # checks if child is subclass of parent
print(issubclass(parent,child))      # checks if parent is subclass of child
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options
Python 3.10.0 (tags/v3.10.0-alpha-0-20210501 | AMD64) [on win32]
Type "help", "copyright" or "credits" for more information.
>>> ===== RESTART: D:\PycharmProjects\Python\Day 10\Inheritance.py
True
False
>>>
```

Inheritance



Isinstance():

- Isinstance() is another inbuilt function of python that allows us to check whether an object is an instance of a particular class or any of the classes it has been derived from.
- It takes two parameters the object and the class we need to check it against.

Example:

```
# isinstance() example

class parent:           # parent class
    def func1():
        print("Hello Parent")

class child(parent):    # child class
    def func2():
        print("Hello Child")

A = child()            # objects initialized
B = parent()

print(isinstance(A,child))      # checks if A is instance of child
print(isinstance(A,parent))     # checks if A is instance of parent
print(isinstance(B,child))      # checks if B is instance of child
print(isinstance(B,parent))    # checks if B is instance of parent
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug
Python 3.10.0 (AMD64) on win3
Type "help", "c
>>>
=====
True
True
False
True
>>> |
```

Thank You



Abstract Class

Abstract Class



What is Abstract Class?

- An Abstract class can be considered as a blueprint for other classes.
- It allows us to create a set of methods that must be created within any child classes built from the abstract class.
- An abstract method is a method that has a declaration but does not have an implementation.
- When we want to provide a common interface for different implementations of a component, we use an abstract class.

Abstract Class



Why we use Abstract Class:

- By defining an abstract class, we can define a common Application Program Interface(API) for a set of subclasses.
- This capability is especially useful in situations where a third – party is going to provide implementations, such as with plugins.
- It can also help you when working in a large team or with a large code-base where keeping all classes in your mind is difficult or not possible.

Abstract Class



How Abstract Classes work:

- By default, Python does not provide abstract classes.
- Python comes with a module that provides the base for defining Abstract Base Classes(ABC) and that module name is ABC.
- ABC works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base.
- A method becomes abstract when decorated with the `@abstractmethod` keyword.

Abstract Class



Example:

The Employee class:

```
from abc import ABC, abstractmethod

class Employee(ABC):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"

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

Abstract Class



Example:

The FulltimeEmployee class:

```
class FulltimeEmployee(Employee):

    def __init__(self, first_name, last_name, salary):
        super().__init__(first_name, last_name)
        self.salary = salary

    def get_salary(self):
        return self.salary
```

Abstract Class



Example:

The *HourlyEmployee* class:

```
class HourlyEmployee(Employee):

    def __init__(self, first_name, last_name, worked_hours, rate):
        super().__init__(first_name, last_name)
        self.worked_hours = worked_hours
        self.rate = rate

    def get_salary(self):
        return self.worked_hours * self.rate
```

Abstract Class



Example:

The Payroll class:

```
class Payroll:
    def __init__(self):
        self.employee_list = []

    def add(self, employee):
        self.employee_list.append(employee)

    def print(self):
        for e in self.employee_list:
            print(f"{e.full_name} \t ${e.get_salary()}")
```

Abstract Class



Example:

The Main program file:

```
from fulltimeemployee import FulltimeEmployee
from hourlyemployee import HourlyEmployee
from payroll import Payroll

payroll = Payroll()

payroll.add(FulltimeEmployee('John', 'Doe', 6000))
payroll.add(FulltimeEmployee('Jane', 'Doe', 6500))
payroll.add(HourlyEmployee('Jenifer', 'Smith', 200, 50))
payroll.add(HourlyEmployee('David', 'Wilson', 150, 100))
payroll.add(HourlyEmployee('Kevin', 'Miller', 100, 150))

payroll.print()
```

Abstract Class



Output:

John Doe	\$6000
Jane Doe	\$6500
Jenifer Smith	\$10000
David Wilson	\$15000
Kevin Miller	\$15000

Thank You



Abstraction

Abstraction



What is Abstraction?

- Data abstraction is the process of hiding implementational details and showing only functionality to the user.
- It shows only the essential things to the user and hides the internal details.
- Abstraction is done by using Abstract keyword.

Abstraction



Algorithm To Implement Abstraction:

- Determine the classes or interfaces that will be part of the abstraction.
- Create an abstract class or interface that defines the common behaviors and properties of these classes.
- Define abstract methods within the abstract class or interface that do not have any implementations details.
- Implement concrete classes that extend the abstract class or implement the interface.
- Override the abstract methods in the concrete classes to provide their specific implementations.
- Use the concrete classes to implement the program logic.

Abstraction Using Abstract Class



Example:

```
# abstraction in python
from abc import ABC, abstractmethod

# abstract class
class Subject(ABC):
    @abstractmethod
    def subject(self):
        pass

class Maths(Subject):
    # override superclass method
    def subject(self):
        print("Subject is Maths")

class Physics(Subject):
    # override superclass method
    def subject(self):
        print("Subject is Physics")

class Chemistry(Subject):
    # override superclass method
    def subject(self):
        print("Subject is Chemistry")

class English(Subject):
    # override superclass method
    def subject(self):
        print("Subject is English")

maths=Maths()
maths.subject()

physics=Physics()
physics.subject()

chemistry=Chemistry()
chemistry.subject()

english=English()
english.subject()
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window
Python 3.10.0 (tags/v3.10.0:k
AMD64) on win32
Type "help", "copyright", "cr
>>> ===== RESTART: D:/Syllak
Subject is Maths
Subject is Physics
Subject is Chemistry
Subject is English
>>> |
```

Thank You



Interface

Interface



Interface:

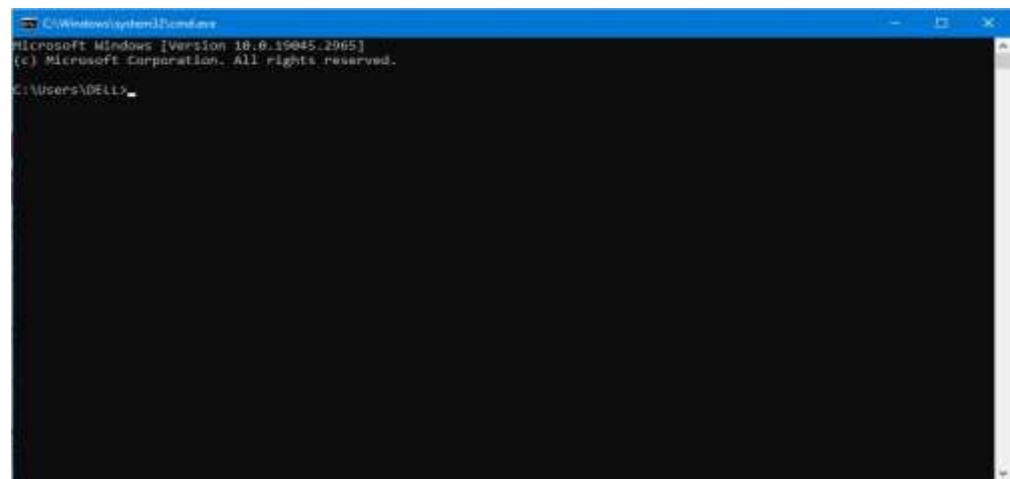
- The interface is a collection of method signatures that should be provided by the implementing class.
- Implementing an interface is a way of writing an organized code and achieve abstraction.
- The package Zope.interface provides an implementation of “Object interfaces” for python.
- The package exports two objects, ‘Interface’ and ‘Attribute’ directly.

Interface



Steps to Install zope package:

Step1: Open command prompt.



Step2: Run the following command.

```
C:\Users\DELL>pip install zope
```

Interface



Steps to Install zope package:

Step3: After installation it shows successfully installed zope-5.8.2.

```
C:\Windows\system32\cmd.exe
ope) (3.0.post1)
Requirement already satisfied: zodbpickle>=1.0.1 in c:\users\dell\appdata\roaming\python\python310\site-packages (from Z
ODB->zope) (3.0.1)
Requirement already satisfied: zope.dottedname in c:\users\dell\appdata\roaming\python\python310\site-packages (from zop
e.container->zope) (6.0)
Requirement already satisfied: zope.cachedescriptors in c:\users\dell\appdata\roaming\python\python310\site-packages (fr
om zope.container->zope) (5.0)
Requirement already satisfied: zope.filerepresentation in c:\users\dell\appdata\roaming\python\python310\site-packages (
from zope.container->zope) (6.0)
Requirement already satisfied: zope.annotation in c:\users\dell\appdata\roaming\python\python310\site-packages (from zop
e.site->zope) (5.0)
Requirement already satisfied: BeautifulSoup4 in c:\users\dell\appdata\roaming\python\python310\site-packages (from zope
.testbrowser->zope) (4.12.2)
Requirement already satisfied: WebTest>=2.0.30 in c:\users\dell\appdata\roaming\python\python310\site-packages (from zop
e.testbrowser->zope) (3.0.0)
Requirement already satisfied: WSGIProxy2 in c:\users\dell\appdata\roaming\python\python310\site-packages (from zope.tes
tbrowser->zope) (0.5.1)
Requirement already satisfied: SoupSieve>=1.9.0 in c:\users\dell\appdata\roaming\python\python310\site-packages (from zo
pe.testbrowser->zope) (2.4.1)
Requirement already satisfied: WebOb>=1.2 in c:\users\dell\appdata\roaming\python\python310\site-packages (from WebTest>
=2.0.30->zope.testbrowser->zope) (1.8.7)
Installing collected packages: zope
  WARNING: The scripts addzopeuser.exe, mkwsgiinstance.exe, runwsgi.exe and zconsole.exe are installed in 'C:\Users\DELL
\AppData\Roaming\Python\Python310\Scripts' which is not on PATH.
    Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed zope-5.8.2
WARNING: You are using pip version 21.2.3; however, version 23.1.2 is available.
You should consider upgrading via the 'C:\Program Files\Python310\python.exe -m pip install --upgrade pip' command.
C:\Users\DELL>
```

Interface



Declaring Interface:

- In python, Interface is defined using python class statements and is a subclass of `interface.Interface` which is the parent interface for all interfaces.

Syntax:

```
class IMyInterface(zope.interface.Interface):  
    # methods and attributes
```

Interface



Example:

```
import zope.interface

class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute("foo")
    def method1(self, x):
        pass
    def method2(self):
        pass

print(type(MyInterface))
print(MyInterface.__module__)
print(MyInterface.__name__)

# get attribute
x = MyInterface['x']
print(x)
print(type(x))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021,
AMD64) ] on win32
Type "help", "copyright", "credits" or "license()"
>>>
=====
RESTART: D:\Syllabus\Python\Example\E
<class 'zope.interface.interface.InterfaceClass'>
__main__
MyInterface
__main__.MyInterface.foo
<class 'zope.interface.interface.Attribute'>
>>> |
```

Interface



Implementing Interface:

- Interface acts as a blueprint for designing classes, so interfaces are implemented using implementer decorator on class.
- If a class implements an interface, then the instances of the class provide the interface.
- Objects can provide interfaces directly in addition to what their classes implement.

Interface



Syntax:

```
@zope.interface.implementer(*interfaces)
class Class_name:
    # methods
```

Example:

```
import zope.interface

class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute("foo")
    def method1(self, x):
        pass
    def method2(self):
        pass

@zope.interface.implementer(MyInterface)
class MyClass:
    def method1(self, x):
        return x**2
    def method2(self):
        return "foo"
```

Interface



Implementation Methods:

- *implementedBy(class)* – Returns a boolean value. True if class implements the interface else False.
- *providedBy(object)* – Returns a boolean value, True if object provides the interface else False.
- *providedBy(class)* – Returns False as class does not provide interface but implements it.
- *List(zope.interface.implementedBy(class))* – Returns the list of interfaces implemented by a class.

Interface



Implementation Methods:

- *List(zope.interface.providedBy(object))* – Returns the list of interfaces provided by an object.
- *List(zope.interface.providedBy(class))* – Returns empty list as class does not provide interface but implements it.

Interface



Example:

```
class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute('foo')
    def method1(self, x, y, z):
        pass
    def method2(self):
        pass

@zope.interface.implementer(MyInterface)
class MyClass:
    def method1(self, x):
        return x**2
    def method2(self):
        return "foo"
obj = MyClass()

# ask an interface whether it
# is implemented by a class:
print(MyInterface.implementedBy(MyClass))

# MyClass does not provide
# MyInterface but implements it:
print(MyInterface.providedBy(MyClass))

# ask whether an interface
# is provided by an object:
print(MyInterface.providedBy(obj))

# ask what interfaces are
# implemented by a class:
print(list(zope.interface.implementedBy(MyClass)))

# ask what interfaces are
# provided by an object:
print(list(zope.interface.providedBy(obj)))

# class does not provide interface
print(list(zope.interface.providedBy(MyClass)))
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct
AMD64)] on win32
Type "help", "copyright", "credits" or "l
>>>
=====
RESTART: D:\Syllabus\Python
True
False
True
[<InterfaceClass interface.MyInterface>]
[<InterfaceClass interface.MyInterface>]
[]
>>> |
```

Interface



Interface Inheritance:

- Interfaces can extend other interfaces by listing the other interfaces as base interfaces.

Functions:

- *Extends(interface)* – returns boolean value, whether one interface extends another.
- *isOrExtends(interface)* – returns boolean value, whether interfaces are same or one extends another.
- *isEqualOrExtendedBy(interface)* – Returns boolean value, whether interfaces are same or one is extended by another.

Interface



```
import zope.interface

Example: class BaseI(zope.interface.Interface):
    def m1(self, x):
        pass
    def m2(self):
        pass

    class DerivedI(BaseI):
        def m3(self, x, y):
            pass

    @zope.interface.implementer(DerivedI)
    class cls:
        def m1(self, z):
            return z***3
        def m2(self):
            return 'foo'
        def m3(self, x, y):
            return x ^ y

    # Get base interfaces
    print(DerivedI.__bases__)

    # Ask whether baseI extends
    # DerivedI
    print(BaseI.extends(DerivedI))

    # Ask whether baseI is equal to
    # or is extended by DerivedI
    print(BaseI.isEqualOrExtendedBy(DerivedI))

    # Ask whether baseI is equal to
    # or extends DerivedI
    print(BaseI.isOrExtends(DerivedI))

    # Ask whether DerivedI is equal
    # to or extends BaseI
    print(DerivedI.isOrExtends(DerivedI))
```

Output:

```
(<InterfaceClass __main__.BaseI>, )
False
True
False
True
```

Interface



Interface VS Abstract class:

Python interface	Python abstract class
An interface is a set of methods and attributes on that object.	We can use an abstract base class to define and enforce an interface.
All methods of an interface are abstract	An abstract class can have abstract methods as well as concrete methods.
We use an interface if all the features need to be implemented differently for different objects.	Abstract classes are used when there is some common feature shared by all the objects as they are.
The interface is slow as compared to the abstract class.	Abstract classes are faster.

Thank You



String Concatenation

String Concatenation



String Concatenation:

- String Concatenation is the technique of combining two strings.
- Strings are immutable, therefore, whenever it is concatenated, it is assigned to a new variable.

This can be done by using many ways, They are:

- Using + Operator
- Using join() Method
- Using % Operator
- Using format() function
- Using comma(,).

String Concatenation



String Concatenation Using + Operator:

- It's very easy to use the + operator for string concatenation.
- This operator can be used to add multiple strings together.

String Concatenation



Example:

```
# Defining strings
var1 = "Hello "
var2 = "World"

# + Operator is used to combine strings
var3 = var1 + var2
print(var3)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window
Python 3.10.0 (tags/v3.10.0:b
AMD64) on win32
Type "help", "copyright", "cr
>>>
===== RESTART: D:/Syllabus/P
Hello World
>>> |
```

String Concatenation



String Concatenation Using join() Method:

- The Join() method is a string method and returns a string in which the elements of the sequence have been joined by str separator.
- This method combines the string that is stored in the var1 and var2.
- It accepts only the list as its argument and list size can be anything.

String Concatenation



Example:

```
var1 = "Hello"
var2 = "World"

# join() method is used to combine the strings
print("".join([var1, var2]))

# join() method is used here to combine
# the string with a separator Space(" ")
var3 = " ".join([var1, var2])

print(var3)
```

Output:

```
IDLE Shell 3.10.0
File Edit Shell Debug Options Window
Python 3.10.0 (tags/v3.10.0b2
AMD64) ] on win32
Type "help", "copyright", "c:
>>>
===== RESTART: D:/Syllabus/1
HelloWorld
Hello World
>>>
```

String Concatenation



String Concatenation Using % Operator:

- We can use the % operator for string formatting.
- It's useful when we want to concatenate strings and perform simple formatting.
- The %s denotes string data type.

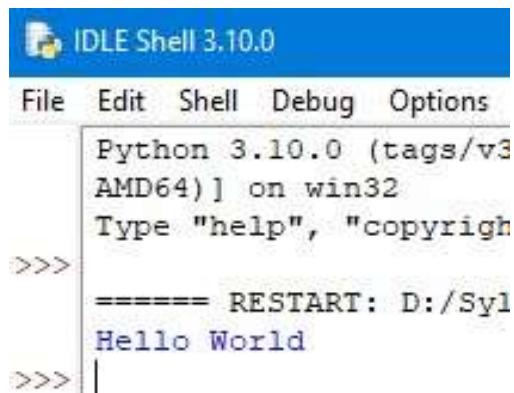
String Concatenation



Example:

```
var1 = "Hello"  
var2 = "World"  
  
# % Operator is used here to combine the string  
print("% s % s" % (var1, var2))
```

Output:



```
IDLE Shell 3.10.0  
File Edit Shell Debug Options  
Python 3.10.0 (tags/v3  
AMD64) ] on win32  
Type "help", "copyright"  
>>>  
===== RESTART: D:/Syl  
Hello World  
>>> |
```

Thank You



File Handling

File Handling



What is File Handling?

- File Handling in python is a powerful and versatile tool that can be used to perform a wide range of operations.
- The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy.
- But, like other concepts file handling in python is also easy and short.
- Each line of code includes a sequence of characters and they form a text file.
- Each line of a file is terminated with a special character called EOL.
- EOL(End Of Line) ends the current line and tells the interpreter a new one has begun.

File Handling



Advantages of File Handling:

Versatility:

- File handling in python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming and deleting files.

Flexibility:

- File handling in python is highly flexible. As it allows you to work with different file types (e.g: text files, binary files, CSV files etc) and to perform different operations on files(e.g: read, write, append etc)

File Handling



Advantages of File Handling:

User-friendly:

- Python provides a user – friendly interface for file handling, making it easy to create, read and manipulate files.

Cross-platform:

- Python file – handling functions work across different platforms(e.g. windows, Mac, Linux) allowing for seamless integration and compatibility.

File Handling



Disadvantages of File Handling:

Error-prone:

- File handling operations in python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g file permissions, file locks, etc).

Security risks:

- File handling in python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.

File Handling



Disadvantages of File Handling:

Complexity:

- File handling in python can be complex, especially when working with more advanced file formats or operations.
- Careful attention must be paid to the code to ensure that files are handled properly and securely.

Performance:

- File handling operations in python can be slower than other programming languages. Especially when dealing with large files or performing complex operations.

File Handling



Open() function:

- Before performing any operations on the file like reading or writing. First, we have to open that file.
- For this, we should use python's inbuilt function `open()`.
- At the time of opening, we have to specify the mode, which represents the purpose of the opening file.

Syntax:

```
f = open(filename, mode)
```

File Handling



Modes in open() function:

- ***r:*** Open an existing file for a read operation.
- ***w:*** Open an existing file for a write operation. If the file already contains some data then it will be overridden but if the file is not present then it creates the file as well.
- ***a:*** Open an existing file for append operation. It won't override existing data.
- ***r+:*** To read and write data into the file. The previous data in the file will be overridden.
- ***w+:*** To write and read data. It will override existing data.
- ***a+:*** To append and read data from the file. It won't override existing data.

File Handling



Working in Write mode:

- The write method is used to write user contents in an file.
- If the file already contains some data then it will be overridden.
- But if the file is not present then it creates the file.

Example:

```
file = open('welcome.txt', 'w')
file.write("Hi welcome to file handling")
file.write("this is the write method")
file.close()
```

Output:

NAME	DATE CREATED	TYPE	SIZE
fileHandling	7/24/2023 3:33 PM	Python File	1 KB
welcome	7/24/2023 3:33 PM	Text Document	1 KB

File Handling



Working in Read mode:

- The Read method is used to read user contents from an file.

Method 1:

- The open command will open the file in the read mode and the for loop will print each line present in the file.

Example:

```
# a file named "welcome", will be opened with the reading mode.
file = open('welcome.txt', 'r')

# This will print every line one by one in the file
for each in file:
    print (each)
```

Output:

```
===== RESTART: C:\Users\DELL\Desktop\File handling\fileHandling.py =====
Hi welcome to file handlingthis is the write method
```

File Handling



Working in Read mode:

- The Read method is used to read user contents from an file.

Method 2:

- In this method, we will extract a string that contains all characters in the file then we can use file.read().

Example:

```
# a file named "welcome", will be opened with the reading mode.  
file = open('welcome.txt', 'r')  
# Python code to illustrate read() mode  
print (file.read())
```

Output:

```
===== RESTART: C:\Users\DELL\Desktop\File handling\fileHandling.py  
Hi welcome to file handlingthis is the write method
```

File Handling



Working in Read mode:

- The Read method is used to read user contents from an file.

Method 3:

- Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string.

Example:

```
# Python code to illustrate read() mode character wise
file = open("welcome.txt", "r")
print (file.read(5))
```

Output:

```
===== RESTART: C:\Users\DEL
Hi we
```

File Handling



Working in Read mode:

- The Read method is used to read user contents from an file.

Method 4:

- We can also split lines while reading files in python.
- The split() function splits the variable when space is encountered. You can also split using any characters as you wish.

Example:

```
# Python code to illustrate split() function
with open("welcome.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

Output:

```
===== RESTART: C:\Users\DELL\Desktop\File handling\fileHandling.py =====
['Hi', 'welcome', 'to', 'file', 'handlingthis', 'is', 'the', 'write', 'method']
```

File Handling



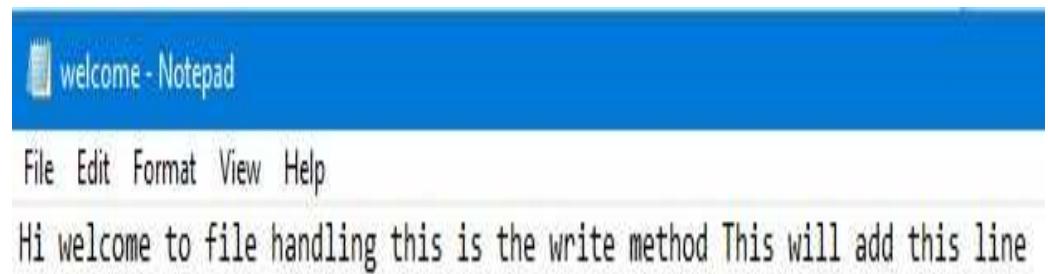
Working in Append mode:

- The Append method is used to add additional content to an existing file.

Example:

```
# Python code to illustrate append() mode
file = open('welcome.txt', 'a')
file.write("This will add this line")
file.close()
```

Output:



Thank You



Map() function

Map() Function



Map() function:

- The map() function is used to apply a function to each element of the Iterable and returns a new Iterable with the outcomes.
- The map() function takes two arguments: a function and an Iterable.
- The function contention is the function that will be applied to every element of the Iterable.
- The Iterable contention is the Iterable that the function will be applied to.

Map() Function



Syntax:

```
map(function, iterables)
```

Parameters:

Function: It is a function in which a map passes each item of the Iterable.

Iterable: It is a sequence, collection or an iterator object which is to be mapped.

Map() Function



Example1:

- Utilizing map() to square a list of numbers

Code:

```
# Python program to demonstrate working
# of map.

# Return square of n

def square(n):
    return n*n

# We double all numbers using map()
numbers = (30, 2, 3, 4)
result = map(square, numbers)
print(list(result))
```

Output:

```
=====
RESTART: C:/Users/DELL/Desktop/map.py =====
[900, 4, 9, 16]
```



Map() Function

Example1:

- Utilizing map() to square a list of numbers

Code:

```
# Python program to demonstrate working
# of map.

# Return square of n

def square(n):
    return n*n

# We double all numbers using map()
numbers = (30, 2, 3, 4)
result = map(square, numbers)
print(list(result))
```

Output:

```
=====
RESTART: C:/Users/DELL/Desktop/map.py =====
[900, 4, 9, 16]
```

Map() Function



Example2:

Utilizing map() without using functions.

Code:

```
# List of strings
l = ['sat', 'bat', 'cat', 'mat']

# map() can listify the list of strings individually
test = list(map(list, l))
print(test)
```

Output:

```
=====
RESTART: C:/Users/DELL/Desktop/map.py =====
[['s', 'a', 't'], ['b', 'a', 't'], ['c', 'a', 't'], ['m', 'a', 't']]
```

Map() Function



Example3:

Utilizing map() with if statement.

Code:

```
# Define a function that doubles even numbers and leaves odd numbers as is
def double_even(num):
    if num % 2 == 0:
        return num * 2
    else:
        return num

# Create a list of numbers to apply the function to
numbers = [1, 2, 3, 4, 5]

# Use map to apply the function to each element in the list
result = list(map(double_even, numbers))

# Print the result
print(result)
```

Output:

```
===== RESTART: C:\Users\DELL\Desktop\map.py =====
[1, 4, 3, 8, 5]
```

Thank You



List comprehension

List Comprehension



List comprehension:

- List comprehension in python is an easy and compact syntax for creating a list from a string or another list.
- It is very concise way to create a new list by performing an operation on each item in the existing list.
- List comprehension is considerably faster than processing a list using the for loop.

Syntax:

```
newList=[expression(element) for element in oldList if condition]
```

List comprehension



Key points to remember:

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating list.
- However, we should avoid writing very long list comprehension in one line to ensure that code is user – friendly.
- Remember, every list comprehension can be rewritten in for loop, but every for loop can't be rewritten in the form of list comprehension.

List Comprehension



Advantages of List comprehension:

- More time- efficient and space-efficient than loops.
- Require fewer lines of codes.
- Transforms iterative statement into a formula.

List Comprehension



Example1:

- Iteration with list comprehension.

Code:

```
# Using list comprehension to iterate through loop
List = [character for character in [1, 2, 3]]

# Displaying list
print(List)
```

Output:

```
===== RESTART: C:/Users/DELL
[1, 2, 3]
```

List Comprehension



Example2:

- Even list using list comprehension.

Code:

```
list = [i for i in range(11) if i % 2 == 0]
print(list)
```

Output:

```
=====
      RESTART:
=====
[0, 2, 4, 6, 8, 10]
```

List Comprehension



List comprehension vs for loop:

For loop:

```
# Empty list
List = []

# Traditional approach of iterating
for character in 'Hello guys....!':
    List.append(character)

# Display list
print(List)
```

Output:

List Comprehension:

```
# Using list comprehension to iterate through loop
List = [character for character in 'Hello guys....!']

# Displaying list
print(List)
```

```
===== RESTART: C:\Users\DELL\Desktop\listComprehension.py =====
['H', 'e', 'l', 'l', 'o', ' ', 'g', 'u', 'y', 's', '.', '.', '.', '!']
```

List Comprehension



Time analysis in list comprehension and for loop:

Example:

```
# Import required module
import time

# define function to implement for loop
def for_loop(n):
    result = []
    for i in range(n):
        result.append(i**2)
    return result

# define function to implement list comprehension
def list_comprehension(n):
    return [i**2 for i in range(n)]

# Driver Code

# Calculate time taken by for_loop()
begin = time.time()
for_loop(10**6)
end = time.time()

# Display time taken by for_loop()
print('Time taken for_loop:', round(end-begin, 2))

# Calculate time taken by list_comprehension()
begin = time.time()
list_comprehension(10**6)
end = time.time()

# Display time taken by list_comprehension()
print('Time taken for list_comprehension:', round(end-begin, 2))
```

Output:

```
=====
RESTART: C:\Users\DELL\Desktop\Python\ListComprehension.py
Time taken for_loop: 0.61
Time taken for list_comprehension: 0.44
```

Thank You



Iterator

Iterator



Iterator:

- Iterators are methods that iterate collections like list, tuple etc.
- Using an iterator method, we can loop through an object and return its elements.
- A python iterator object must implement two special methods `__iter__()` and `__next__()`, collectively called the iterator protocol.

Iterator



Iterating Through Iterator:

- In python, we can use the `next()` function to return the next item in the sequence.

Example:

```
# define a list
my_list = [4, 7, 0]

# create an iterator from the list
iterator = iter(my_list)

# get the first element of the iterator
print(next(iterator)) # prints 4

# get the second element of the iterator
print(next(iterator)) # prints 7

# get the third element of the iterator
print(next(iterator)) # prints 0
```

Output:

```
=====
4
7
0
```

Iterator



Working of for loops for iterators:

Example:

```
# create a list of integers
my_list = [1, 2, 3, 4, 5]

# create an iterator from the list
iterator = iter(my_list)

# iterate through the elements of the iterator
for element in iterator:
    if element == 3:
        break
    # Print each element
    print(element)

print(next(iterator))
```

Output:

```
=====
1
2
4
```

Iterator



Building custom iterators:

- Building an iterator from scratch is easy in python. We just have to implement the `__iter__()` and the `__next__()` methods.
- `__iter__()` returns the iterator object itself. If required, some initialization can be performed.
- `__next__()` must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise `StopIteration`.

Iterator



Example:

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max=0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration

# create an object
numbers = PowTwo(3)

# create an iterable from the object
i = iter(numbers)

# Using next to get to the next iterator element
print(next(i)) # prints 1
print(next(i)) # prints 2
print(next(i)) # prints 4
print(next(i)) # prints 8
print(next(i)) # raises StopIteration exception
```

Output:

```
===
1
2
4
8
Traceback (most recent call last):
  File "C:\Users\DELL\Desktop\iter.py", line 32, in <module>
    print(next(i)) # raises StopIteration exception
  File "C:\Users\DELL\Desktop\iter.py", line 18, in __next__
    raise StopIteration
StopIteration
```

Thank You



Generator

Generator



Generator:

- A generator in python is a function that returns an iterator using the Yield keyword.
- A generator function in python is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return.
- If the body of a def contains yield, the function automatically becomes a Python generator function.

Generator



Create a Generator:

- We can create a generator function by simply using the def keyword and the yield keyword.

Syntax:

```
def function_name():  
    yield statement
```

Generator



Example:

```
# A generator function that yields 1 for first time,  
# 2 second time and 3 third time  
a = 2  
b = 3  
  
def simpleGeneratorFun(x,y):  
    yield x+y  
    yield x-y  
    yield x*y  
  
# Driver code to check above generator function  
for value in simpleGeneratorFun(a,b):  
    print(value)
```

Output:

```
===== RESTART: C:/Users/DELL/Desktop/generator.py  
5  
-1  
6
```

Generator



Generator object:

- Python Generator functions return a generator object that is Iterable.
- Generator objects are used either by calling the next method of the generator object or using the generator object in a “for in ” loop.

Generator



Example:

```
# A Python program to demonstrate use of
# generator object with next()

# A generator function
def simpleGeneratorFun():
    yield "Hi"
    yield "welcome"
    yield "generator"

# x is a generator object
x = simpleGeneratorFun()

# Iterating over the generator object using next()

# In Python 3, __next__()
print(next(x))
print(next(x))
print(next(x))
```

Output:

```
=====
RESTART: C:/Users,
Hi
welcome
generator
>
```

Generator



Example:

- In this example we will create two generators for Fibonacci Numbers, first a simple generator and second generator using for loop.

Code:

```
# A simple generator for Fibonacci Numbers
def fib(limit):

    # Initialize first two Fibonacci Numbers
    a, b = 0, 1

    # One by one yield next Fibonacci Number
    while a < limit:
        yield a
        a, b = b, a + b

# Create a generator object
x = fib(5)

# Iterating over the generator object using next
# In Python 3, __next__()
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))

# Iterating over the generator object using for
# in loop.
print("\nUsing for in loop")
for i in fib(5):
    print(i)
```

Output:

```
=====
RESTART:
0
1
1
2
3
```

Using for in loop

```
0
1
1
2
3
```

Generator



Generator Expression:

- Generator Expression is another way of writing the generator function.
- It uses the python list comprehension technique.
- Instead of storing the elements in a list memory, It creates a generator objects.

Syntax:

(expression for item in iterable)

Generator



Example:

```
# generator expression
generator_exp = (i * 5 for i in range(5) if i%2==0)

for i in generator_exp:
    print(i)
```

Output:

```
===== RES
0
10
20
```

Generator



Advantages:

- Easier to built iterators using generators.
- They are memory efficient since they produce one item at a time.
- They can represent an infinite stream of data.

Generator



Disadvantages:

- Every time you want to reuse the elements in a collection it must be regenerated.
- Your code could be more trickier to read when performing lazy evaluation.

Thank You



Iertools

Itertools



Itertools:

- Itertools is a module in python, it is used to iterate over data structures that can be stopped over using a for-loop.
- Such data structures are also known as iterables.
- This modules works as a fast, memory-efficient tool that is used either by themselves or in combination to form iterator algebra.

Types of Iterators:

- Infinite Iterators
- Combinatoric iterators
- Terminating iterators

Itertools



Infinite Iterator:

- In python, any object that can implement for loop is called iterators.
- Iterator can also be infinite and this type of iterator is called infinite iterator.

Iterator	Argument	Results
count(start,step)	start, [step]	start, start+step, step+2*step
cycle()	P	p0,p1,...,plast
repeat()	elem [,n]	elem, elem, elem,...endlessly or upto n times

Itertools



○ ***count(start, step):***

- It prints from the start value to infinite.
- The step argument is optional, if the value is provided to the step the number of steps will be skipped.

Example:

```
import itertools
for i in itertools.count(10,5):
    if i == 50:
        break
    else:
        print(i,end=" ")
```

Output:

```
===== RESTART: C:
10 15 20 25 30 35 40 45
```

Itertools



○ *cycle(Iterable):*

- This iterator prints all the value in sequence from the passed argument.
- It prints values In a cycle manner.

Example:

```
import itertools
temp = 0
for i in itertools.cycle("123"):
    if temp > 7:
        break
    else:
        print(i, end=' ')
        temp = temp+1
```

Output:

```
=====
1 2 3 1 2 3 1 2
```

Itertools



○ *repeat(value, num):*

- As the name suggests, it repeatedly prints the passed value for infinite time.
- The num argument is used to define the repetition time.

Example:

```
import itertools
print("Printing the number repeatly:")
print(list(itertools.repeat(40,15)))
```

Output:

Printing the number repeatly:

[40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40]

Combinatoric Iterators:

- The complex combinatorial constructs are simplified by the recursive generators.
- The permutations, combinations and Cartesian products are the example of the combinatoric construct.

Types:

- Product()
- Permutations()
- Combinations()
- Combination_with_replacement()

Product():

- It is used to calculate the cartesian product of input Iterable.
- In this function, we use the optional repeat keyword argument for computation of the product of an Iterable with itself.
- The repeat keyword represents the number of repetitions.
- It returns output in the form of sorted tuples.

Itertools



Example:

```
from itertools import product

print("We are computing cartesian product using repeat Keyword Argument:")
print(list(product([5, 2], repeat=2)))
print()

print("We are computing cartesian product of the containers:")
print(list(product(['hi', 'welcoome', 'soldier'], '5')))
print()

print("We are computing product of the containers:")
print(list(product('CD', [4, 5])))
```

Output:

```
We are computing cartesian product using repeat Keyword Argument:  
[(5, 5), (5, 2), (2, 5), (2, 2)]
```

```
We are computing cartesian product of the containers:  
[('hi', '5'), ('welcome', '5'), ('soldier', '5')]
```

```
We are computing product of the containers:  
[('C', 4), ('C', 5), ('D', 4), ('D', 5)]
```

Permutations():

- It is used to generate all possible permutation of an Iterable.
- The uniqueness of each element depends upon their position instead of values.
- It accepts two arguments Iterable and group size.
- If the value of group size is none or not specified then group size turns into length of the Iterable.

Itertools



Example:

```
# Python code to demonstrate permutations
import itertools as it

print(list(it.permutations(['d', 'o', 'g'])))
```

Output:

```
----- RESTART: C:\Users\DELL\Desktop\Itertools.py -----
[('d', 'o', 'g'), ('d', 'g', 'o'), ('o', 'd', 'g'), ('o', 'g', 'd'), ('g', 'd', 'o'), ('g', 'o', 'd')]
```

Itertools



Combinations():

- It is used to print all the possible combinations (without replacement) of the container which is passed as argument in the specified group size in sorted order.

Example:

```
# Python code to demonstrate combinations
import itertools as it

print(list(it.combinations([1, 2, 3, 4, 5], 2)))
```

Output:

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

Combination_with_replacement():

- It accepts two arguments, first argument is a r-length tuple and the second argument is repetition.
- It returns a subsequence of length n from the elements of the Iterable and repeat the same process. Separate elements may repeat itself in Combination_with_replacement().

Example:

```
# Python code to demonstrate combinations_with_replacement
import itertools as it
print(list(it.combinations_with_replacement([1, 2, 3, 4, 5], 2)))
```

Output:

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

Terminating Iterator:

- Terminating iterators are generally used to work on the small input sequence and generate the output based on the functionality of the method used in iterator.

Types:

- Accumulate()
- Chain()
- Islice()
- Starmap()

Accumulate(*iter,func*):

- It takes two arguments, first argument is Iterable and the second is a function which would be followed at each iteration of value in iterable.
- If the function is not defined in accumulate() iterator, addition takes place by default.
- The output iterable depends on the input iterable.
- If input iterable contains no value then the output iterable will also be empty.

Iertools



Example:

```
import itertools
import operator

# initializing list 1
list1 = [1, 4, 5, 7, 9, 11]

# using accumulate() that will prints the successive summation of elements
print("The sum is : ", end="")
print(list(itertools.accumulate(list1)))

# using accumulate() that will prints the successive multiplication of elements
print("The product is : ", end="")
print(list(itertools.accumulate(list1, operator.mul)))

# using accumulate() that will prints the successive summation of elements
print("The sum is : ", end="")
print(list(itertools.accumulate(list1)))

# using accumulate() that will prints the successive multiplication of elements
print("The product is : ", end="")
print(list(itertools.accumulate(list1, operator.mul)))
```

Output:

```
=====
KESIARI: C:\USERS\DELL\DESKTOP
The sum is : [1, 5, 10, 17, 26, 37]
The product is : [1, 4, 20, 140, 1260, 13860]
The sum is : [1, 5, 10, 17, 26, 37]
The product is : [1, 4, 20, 140, 1260, 13860]
```

Itertools



Chain(*iter1,iter2*):

- It is used to print all the values in iterable passed in the form of chain and declared in arguments.

Example:

```
import itertools

# declaring list 1
list1 = [1, 2, 3, 4]

# declaring list 2
list2 = [1, 5, 6, 8]

# declaring list 3
list3 = [9, 10, 11, 12]

# using chain() function that will to print all elements of lists
print("The output is : ", end="")
print(list(itertools.chain(list1, list2, list3)))
```

Output:

```
The output is : [1, 2, 3, 4, 1, 5, 6, 8, 9, 10, 11, 12]
```

Itertools



islice(iterable,start,stop,step):

- It slices the given iterable according to given position.
- It accepts four arguments respectively and these are iterable, container, starting position, ending position and step.

Example:

```
import itertools
# Declaring list
list1 = [12, 34, 65, 73, 80, 19, 20]
# using islice() iterator that will slice the list acc. to given argument
# starts printing from 3rd index till 8th skipping 2
print("The sliced list values are : ", end="")
print(list(itertools.islice(list1, 2, 8, 2)))
```

Output:

```
----- RESTART: C:\Users\DELL\DE
The sliced list values are : [65, 80, 20]
```

Itertools



starmap(func, tuple list):

- It takes two arguments, first argument is function and second argument is list which consists element in the form of tuple.

Example:

```
import itertools

# Declaring list that contain tuple as element
list1 = [(10, 20, 15), (18, 40, 19), (53, 42, 90), (16, 12, 27)]

# using starmap() iterator for selection value acc. to function
# selects max of all tuple values
print("The values acc. to function are : ", end="")
print(list(itertools.starmap(max, list1)))
```

Output:

```
===== RESTART: C:\Users\DELL\Desktop\ite
The values acc. to function are : [20, 40, 90, 27]
```

Itertools



Zip_longest(iterable1, iterable2, fillval):

- It prints the values of iterable alternatively In sequence.
- If one of the iterable prints all values, remaining values are filled by the values assigned to fill value.

Example:

```
import itertools
print(" The combined value of iterables is :")
print(*itertools.zip_longest('python', 'programming', fillvalue='_'))
```

Output:

REPL[Python]:> The combined value of iterables is :

('p', 'p') ('y', '_') ('t', '_') ('h', 'g') ('o', 'r') ('n', 'a') ('_','m')
('_','m') ('_','i') ('_','n') ('_','g')

Thank You



Lambda Function

Lambda Function



Lambda Function:

- Python Lambda Functions are anonymous function means that the function is without a name.
- As we already know that the def keyword is used to define a normal function in python.
- Similarly, the lambda keyword is used to define an anonymous function in python.

Syntax:

lambda arguments : expression

Lambda Function



Points to remember:

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expression in functions.

Lambda Function



Example:

```
str1 = 'Hi Welcome To Lambda'

upper = lambda string: string.upper()
print(upper(str1))
```

Output:

```
HI WELCOME TO LAMBDA
```

Lambda Function



Difference between lambda and normal function:

Example:

```
def cube(y):
    return y*y*y

lambda_cube = lambda y: y*y*y

# using function defined
# using def keyword
print("Using function defined with `def` keyword, cube:", cube(5))

# using the lambda function
print("Using lambda function, cube:", lambda_cube(5))
```

Output:

```
Using function defined with `def` keyword, cube: 125
Using lambda function, cube: 125
```

Lambda Function



Condition checking using lambda function:

Example:

```
Minimum = lambda x, y : x if (x < y) else y  
print('The Smallest number is:', Minimum( 35, 28 ))
```

Output:

```
The Smallest number is: 28
```

Lambda Function



Lambda function with list comprehension:

Example:

```
is_even_list = [lambda arg=x: arg * 10 for x in range(1, 5)]  
  
# iterate on each lambda function  
# and invoke the function to get the calculated value  
for item in is_even_list:  
    print(item())
```

Output:

```
10  
20  
30  
40
```

Lambda Function



Lambda function with Multiple statements:

Example:

Output:

```
List = [[2,3,4],[1, 4, 16, 64],[3, 6, 9, 12]]  
  
# Sort each sublist  
sortList = lambda x: (sorted(i) for i in x)  
  
# Get the second largest element  
secondLargest = lambda x, f : [y[len(y)-2] for y in f(x)]  
res = secondLargest(List, sortList)  
  
print(res)
```

```
[3, 16, 9]
```

Lambda Function



Filter():

- The filter() function in python takes in a function and a list as arguments.
- This offers an elegant way to filter out all the elements of a sequence for which the function returns True.

Example:

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(filter(lambda x: (x % 2 != 0), li))
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

Lambda Function



Lambda function with map():

Example:

```
# Python code to illustrate
# map() with lambda()
# to get double of a list.

li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(map(lambda x: x*2, li))
print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

Lambda Function



Lambda function with map():

Example:

```
# Python code to illustrate
# reduce() with lambda()
# to get sum of a list

from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print(sum)
```

Output:

193

Lambda Function



Lambda function with map():

Example:

```
# python code to demonstrate working of reduce()
# with a lambda function

# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2, ]

# using reduce to compute maximum element from list
print("The maximum element of the list is : ", end="")
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

Output:

```
The maximum element of the list is : 6
```

Thank You



Lambda Function

Lambda Function



Lambda Function:

- Python Lambda Functions are anonymous function means that the function is without a name.
- As we already know that the def keyword is used to define a normal function in python.
- Similarly, the lambda keyword is used to define an anonymous function in python.

Syntax:

lambda arguments : expression

Lambda Function



Points to remember:

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expression in functions.

Lambda Function



Example:

```
str1 = 'Hi Welcome To Lambda'

upper = lambda string: string.upper()
print(upper(str1))
```

Output:

```
HI WELCOME TO LAMBDA
```

Lambda Function



Difference between lambda and normal function:

Example:

```
def cube(y):
    return y*y*y

lambda_cube = lambda y: y*y*y

# using function defined
# using def keyword
print("Using function defined with `def` keyword, cube:", cube(5))

# using the lambda function
print("Using lambda function, cube:", lambda_cube(5))
```

Output:

```
Using function defined with `def` keyword, cube: 125
Using lambda function, cube: 125
```

Lambda Function



Condition checking using lambda function:

Example:

```
Minimum = lambda x, y : x if (x < y) else y  
print('The Smallest number is:', Minimum( 35, 28 ))
```

Output:

```
The Smallest number is: 28
```

Lambda Function



Lambda function with list comprehension:

Example:

```
is_even_list = [lambda arg=x: arg * 10 for x in range(1, 5)]  
  
# iterate on each lambda function  
# and invoke the function to get the calculated value  
for item in is_even_list:  
    print(item())
```

Output:

```
10  
20  
30  
40
```

Lambda Function



Lambda function with Multiple statements:

Example:

```
List = [[2,3,4],[1, 4, 16, 64],[3, 6, 9, 12]]  
  
# Sort each sublist  
sortList = lambda x: (sorted(i) for i in x)  
  
# Get the second largest element  
secondLargest = lambda x, f : [y[len(y)-2] for y in f(x)]  
res = secondLargest(List, sortList)  
  
print(res)
```

Output:

```
[3, 16, 9]
```

Lambda Function



Filter():

- The filter() function in python takes in a function and a list as arguments.
- This offers an elegant way to filter out all the elements of a sequence for which the function returns True.

Example:

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(filter(lambda x: (x % 2 != 0), li))
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

Lambda Function



Lambda function with map():

Example:

```
# Python code to illustrate
# map() with lambda()
# to get double of a list.

li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(map(lambda x: x*2, li))
print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

Lambda Function



Lambda function with map():

Example:

```
# Python code to illustrate
# reduce() with lambda()
# to get sum of a list

from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print(sum)
```

Output:

193

Lambda Function



Lambda function with map():

Example:

```
# python code to demonstrate working of reduce()
# with a lambda function

# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2, ]

# using reduce to compute maximum element from list
print("The maximum element of the list is : ", end="")
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

Output:

```
The maximum element of the list is : 6
```

Thank You



Exception Handling

Exception Handling



Exception Handling:

- An exception In python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted.
- When a python code comes across a condition it can't handle, it raises an exception.
- An object in python that describes an error is called an exception.

Exception Handling



Different types of Exceptions:

SyntaxError:

- This exception is raised when the interpreter encounters a syntax error in the code such as misspelled keyword, a missing colon or an unbalanced parenthesis.

TypeError:

- This exception is raised when an operation or function is applied to an object of the wrong type such as adding a string to an integer.

Exception Handling



Different types of Exceptions:

NameError:

- This exception is raised when a variable or function name is not found in the current scope.

IndexError:

- This exception is raised when an index is out of range for a list, tuple or other sequence types.

KeyError:

- This exception is raised when a key is not found in a dictionary.

Exception Handling



Different types of Exceptions:

ValueError:

- This exception is raised when a function or method is called with an invalid argument or input.
- Such as trying to convert a string to an integer when the string does not represent a valid integer.

AttributeError:

- This exception is raised when an attribute or method is not found on an object. Such as trying to access a non-existent attribute of a class instance.

Exception Handling



Different types of Exceptions:

IOError:

- This exception is raised when an I/O operation such as reading or writing a file, fails due to an input/output error.

ZeroDivisionError:

- This exception is raised when an attempt is made to divide a number by zero.

ImportError:

- This exception is raised when an import statement fails to find or load a module.

Exception Handling



Try and Except Statement:

- Try and Except statement is used to handle these errors within our code in python.
- The try block is used to check some code for errors i.e. the code inside the try block will execute when there is no error in the program.
- Whereas the code inside the except block will execute whenever the program encounters some error in the preceding try block.

Syntax:

```
try:  
    # Some Code  
except:  
    # Executed if error in the  
    # try block
```

Exception Handling



How try() works?

- First, the try clause is executed i.e. the code between try.
- If there is no exception, then only the try clause will run, except clause is skipped.
- If any exception occurs, the try clause will be skipped and except clause will run.
- If any exceptions occurs, but the except clause within the code doesn't handle it. It is passed on to the outer try statements. If the exception is left unhandled, then the execution stops.
- A try statement can have more than one except clause.

Exception Handling



Example1:

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
        print("Yeah ! Your answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")

# Look at parameters and note the working of Program
divide(3, 2)
```

Output:

```
Yeah ! Your answer is : 1
```

Exception Handling



Example2:

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
        print("Yeah ! Your answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")

# Look at parameters and note the working of Program
divide(3, 0)
```

Output:

```
Sorry ! You are dividing by zero
```

Exception Handling



Example3:

```
# code
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
        print("Yeah ! Your answer is :", result)
    except Exception as e:
        # By this way we can know about the type of error occurring
        print("The error is: ",e)

divide(3, "div")
divide(3,0)
```

Output:

```
The error is: unsupported operand type(s) for //: 'int' and 'str'
The error is: integer division or modulo by zero
```

Exception Handling



Finally Keyword:

- 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 exceptions.

Syntax:

```
try:  
    # Some Code  
except:  
    # Executed if error in the  
    # try block  
else:  
    # execute if no exception  
finally:  
    # Some code .....(always executed)
```

Exception Handling



Example:

```
# 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
```

Thank You



Magic Method

Magic Method



Magic Method:

- To add “magic” to the class we create, we can define special methods called “magic methods”.
- For example: The magic methods `__init__` and `__str__` are always wrapped by double underscores from both sides.
- By granting us accessibility to python’s built-in syntax tools, magic methods can improve the structure of our classes.

Magic Method



__new__()method:

- The magic method `__new__()` is called implicitly by the `__init__()` method.
- The new instance returned by the `__new__()` method is initialised.
- To modify the creation of objects in a user-defined class, we must supply a modified implementation of the `__new__()` magic method.
- We need to provide the first argument as the reference to the class whose object is to be created for this static function.

Magic Method



Syntax:

```
class class_name:  
    def __new__(cls, *args, **kwargs):  
        statements  
        .  
        .  
        return super(class_name, cls).__new__(cls, *args, **kwargs)
```

Example:

```
# Python program to  
# demonstrate __new__  
  
# don't forget the object specified as base  
class A(object):  
    def __new__(cls):  
        print("Creating instance")  
        return super(A, cls).__new__(cls)  
  
    def __init__(self):  
        print("Init is called")  
  
A()
```

Output:

```
Creating instance  
Init is called
```

Magic Method



Example:

```
# Python program to
# demonstrate __new__

class A(object):
    def __new__(cls):
        print("Creating instance")

    # It is not called
    def __init__(self):
        print("Init is called")

A()
```

Output:

Creating instance

Magic Method



__add__() method:

- Python `__add__()` function is one of the magic methods in python that returns a new object(third) i.e. the addition of the other two objects.
- It implements the addition operator “+” in python.

Syntax:

Syntax: `obj1.__add__(self, obj2)`

- *`obj1`: First object to add in the second object.*
- *`obj2`: Second object to add in the first object.*

Returns: Returns a new object representing the summation of the other two objects.

Magic Method



Example:

```
class add:  
    def __init__(self, val):  
        self.val = val  
  
    def __add__(self, val2):  
        return GFG(self.val + val2.val)  
  
obj1 = add("Hi we are ")  
obj2 = add("Adding two objects")  
obj3 = obj1 + obj2  
print(obj3.val)
```

Output:

```
Hi we are Adding two objects
```

Magic Method



Example:

```
class add:  
  
    def __init__(self, val):  
        self.val = val  
  
obj1 = add("Hi we are ")  
obj2 = add("Adding two objects")  
obj3 = obj1 + obj2  
print(obj3.val)
```

Output:

```
Exception has occurred: TypeError  
unsupported operand type(s) for +: 'add' and 'add'  
  
File "D:\Syllabus\Python\Full Syllabus\Core Python\Gama\Practice\magicMethod.py", line 9, in <module>  
    obj3 = obj1 + obj2  
TypeError: unsupported operand type(s) for +: 'add' and 'add'
```

Magic Method



__repr__() Method:

- Python `__repr__()` is one of the magic methods that returns a printable representation of an object in python that can be customized or predefined.
- i.e. we can also create the string representation of the object according to our needs.

Syntax:

Syntax: object.__repr__()

- *object: The object whose printable representation is to be returned.*

Return: Simple string representation of the passed object.



Magic Method

Example:

```
# Python program to show how __repr__ magic method works

# Creating a class
class Method:
    # Calling __init__ method and initializing the attributes of the class
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    # Calling the __repr__ method and providing the string to be printed each time instance is printed
    def __repr__(self):
        return f"Following are the values of the attributes of the class Method:\nx = {self.x}\ny = {self.y}\nz = {self.z}"
instance = Method(4, 6, 2)
print(instance)
```

Output:

Following are the values of the attributes of the class Method:

x = 4

y = 6

z = 2

Magic Method



__lt__() method:

- Python __lt__() method is one magic method that is used to define or implement the functionality of the less than operator "<".
- It returns a boolean value according to the condition. i.e. It returns true if $a < b$ where a and b are the objects of the class.

Syntax:

Syntax: __lt__(self, obj)

- *self: Reference of the object.*
- *obj: It is a object that will be compared further with the other object.*

Returns: Returns True or False depending on the comparison.

Magic Method



Example:

```
class GFG:  
    def __init__(self, Marks):  
        self.Marks = Marks  
    def __lt__(self, marks):  
        return self.Marks < marks.Marks  
  
student1_marks = GFG(90)  
student2_marks = GFG(88)  
  
print(student1_marks < student2_marks)  
print(student2_marks < student1_marks)
```

Output:

```
False  
True
```

Magic Method



Example:

```
class GFG:  
    def __init__(self, Marks):  
        self.Marks = Marks  
student1_marks = GFG(90)  
student2_marks = GFG(88)  
  
print(student1_marks < student2_marks)
```

Output:

```
Exception has occurred: TypeError X  
'<' not supported between instances of 'GFG' and 'GFG'  
  
File "D:\Syllabus\Python\Full Syllabus\Core Python\Gama\Practice\magicMethod.py", line 9, in <module>  
    print(student1_marks < student2_marks)  
TypeError: '<' not supported between instances of 'GFG' and 'GFG'
```

Magic Method



__len__() method:

- Python `__len__()` is one of the various magic methods in python programming language.
- It is basically used to implement the `len()` function in python because whenever we call the `len()` function then internally `__len__` magic method is called.
- It finally returns an integer value that is greater than or equal to zero as it represents the length of the object for which it is called.

Syntax:

Syntax: object.__len__()

- *object: It is the object whose length is to be determined.*

Returns: Returns a non negative integer.

Magic Method



Example:

```
class GFG:  
    def __init__(self, a):  
        self.a = a  
    def __len__(self):  
        return len(self.a)  
  
obj = GFG("Hi welcome to Python")  
print(len(obj))
```

Output:

20

Magic Method



Example:

```
class GFG:  
    def __init__(self, a):  
        self.a = a  
  
obj = GFG("Hi welcome to Python")  
print(len(obj))
```

Output:

```
Exception has occurred: TypeError  
object of type 'GFG' has no len()  
  
File "D:\Syllabus\Python\Full Syllabus\Core Python\Gama\Practice\magicMethod.py", line 7, in <module>  
    print(len(obj))  
TypeError: object of type 'GFG' has no len()
```

Thank You



Regular Expression

Regular Expression



Regular Expression:

- A Regular Expressions(RegEx) is a special sequence of characters that uses a search pattern to find a string or set of strings.
- It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub patterns.
- Python provides a re module that supports the use of regex in python.
- Its primary function is to offer a search where it takes a regular expression and a string.

Regular Expression



Example:

```
import re

s = 'hi how are you, welcome to regular expression'

match = re.search(r'welcome', s)

print('Start Index:', match.start())
print('End Index:', match.end())
```

Output:

```
Start Index: 16
End Index: 23
```

Regular Expression



re.findall():

- Return all non-overlapping matches of pattern in string as a list of Strings.
- The string is scanned left – to – right and matches are returned in the order found.

Example:

```
# A Python program to demonstrate working of
# findall()
import re

# A sample text string where regular expression
# is searched.
string = """Hello my Number is 123456789 and
| | | my friend's number is 987654321"""

# A sample regular expression to find digits.
regex = '\d+'

match = re.findall(regex, string)
print(match)

# This example is contributed by Ayush Saluja.
```

Output:

```
[ '123456789', '987654321' ]
```

Regular Expression



re.compile():

- Regular expressions are compiled into pattern objects.
- Which have methods for various operations such as searching for pattern matches or performing string substitutions.

Example:

```
# Module Regular Expression is imported
# using __import__().
import re

# compile() creates regular expression
# character class [a-e],
# which is equivalent to [abcde].
# class [abcde] will match with string with
# 'a', 'b', 'c', 'd', 'e'.
p = re.compile('[a-e]')

#.findall() searches for the Regular Expression
# and return a list upon finding
print(p.findall("aye, said Mr. Gibenson Stark"))
```

Output:

```
['a', 'e', 'a', 'd', 'b', 'e', 'a']
```

Regular Expression



re.split():

- Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.

Example:

```
import re

# Splitting will occurs only once, at
# '12', returned list will have length 2
print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))

# 'Boy' and 'boy' will be treated same when
# flags = re.IGNORECASE
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags=re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

Output:

```
['On ', 'th Jan 2016, at 11:02 AM']
['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']
['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']
```

Regular Expression



re.sub():

- The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string(3rd parameter) and upon finding the substring pattern is replaced by repl(2nd parameter).
- Count checks and maintains the number of times this occurs.

Regular Expression



Example:

```
import re

# string at "Subject" and "Uber". As the CASE has been ignored, using Flag, 'ub' should
# match twice with the string Upon matching,
# 'ub' is replaced by '~*' in "Subject", and
# in "Uber", 'Ub' is replaced.
print(re.sub('ub', '~*', 'Subject has Uber booked already',
| | | flags=re.IGNORECASE))

# Consider the Case Sensitivity, 'Ub' in
# "Uber", will not be replaced.
print(re.sub('ub', '~*', 'Subject has Uber booked already'))

# As count has been given value 1, the maximum
# times replacement occurs is 1
print(re.sub('ub', '~*', 'Subject has Uber booked already',
| | | count=1, flags=re.IGNORECASE))

# 'r' before the pattern denotes RE, \s is for
# start and end of a String.
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',
| | | flags=re.IGNORECASE))
```

Output:

```
S~*ject has ~*er booked already
S~*ject has Uber booked already
S~*ject has Uber booked already
Baked Beans & Spam
```

Regular Expression



re.subn():

- Subn() is similar to sub() in all ways, except in its way of providing output.
- It returns a tuple with a count of the total of replacement and the new string rather than just the string.

Example:

```
import re

print(re.sub('ub', '~~', 'Subject has Uber booked already'))

t = re.subn('ub', '~~', 'Subject has Uber booked already',
           flags=re.IGNORECASE)
print(t)
print(len(t))

# This will give same output as sub() would have
print(t[0])
```

Output:

```
('S~~ject has Uber booked already', 1)
('S~~ject has ~~er booked already', 2)
2
S~~ject has ~~er booked already
```

Regular Expression



re.escape():

- Returns string with all non - alphanumerics backslashed, this is useful if you want match an arbitrary literal string that may have regular expression metacharacters in it.

Example:

```
import re

# escape() returns a string with BackSlash '\\',
# before every Non-Alphanumeric Character
# In 1st case only ' ', is not alphanumeric
# In 2nd case, ' ', caret '^', '-', '[ ]', '\\'
# are not alphanumeric
print(re.escape("This is Awesome even 1 AM"))
print(re.escape("I Asked what is this [a-9], he said \t ^Wow"))
```

Output:

```
This\\ is\\ Awesome\\ even\\ 1\\ AM
I\\ Asked\\ what\\ is\\ this\\ \\[a\\-9\\],\\ he\\ said\\ \\t\\ ^Wow
```

Regular Expression



re.search():

- This method either returns `None` or a `re.MatchObject` contains information about the matching part of the string.
- This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.

Regular Expression



Example:

```
import re
# in the form of Month name followed by day number
regex = r"([a-zA-Z]+) (\d+)"
match = re.search(regex, "I was born on June 24")
if match != None:
    # We reach here when the expression "([a-zA-Z]+) (\d+)"
    # matches the date string.

    # This will print [14, 21), since it matches at index 14
    # and ends at 21.
    print ("Match at index %s, %s" % (match.start(), match.end()))

    # We use group() method to get all the matches and
    # captured groups. The groups contain the matched values.
    # In particular:
    # match.group(0) always returns the fully matched string
    # match.group(1) match.group(2), ... return the capture
    # groups in order from left to right in the input string
    # match.group() is equivalent to match.group(0)

    # So this will print "June 24"
    print ("Full match: %s" % (match.group(0)))

    # So this will print "June"
    print ("Month: %s" % (match.group(1)))

    # So this will print "24"
    print ("Day: %s" % (match.group(2)))

else:
    print ("The regex pattern does not match.")
```

Output:

```
Match at index 14, 21
Full match: June 24
Month: June
Day: 24
```

Thank You



Threading

Threading



Thread:

- A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS(Operating System).
- In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code.
- For simplicity, you can assume that a thread is simply a subset of a process. A thread contains all this information in a TCB(Thread Control Block).

Threading



Thread Control Block:

- *Thread Identifier:* Unique id(TID) is assigned to every new thread.
- *Stack pointer:* Points to the thread's stack in the process. The stage contains the local variables under the thread's scope.
- *Program Counter:* a register that stores the address of the instruction currently being executed by a thread.
- *Thread State:* can be running, ready, waiting, starting or done.
- *Thread's register set:* registers assigned to thread for computations.
- *Parent process pointer:* A pointer to the Process control block (PCB) of the process that the thread lives on.

Threading



Basic function without using thread:

Example:

```
from time import sleep, perf_counter

def task():
    print('Starting a task...')
    sleep(1)
    print('done')

start_time = perf_counter()

task()
task()

end_time = perf_counter()

print(f'It took {end_time - start_time: 0.2f} second(s) to complete.')
```

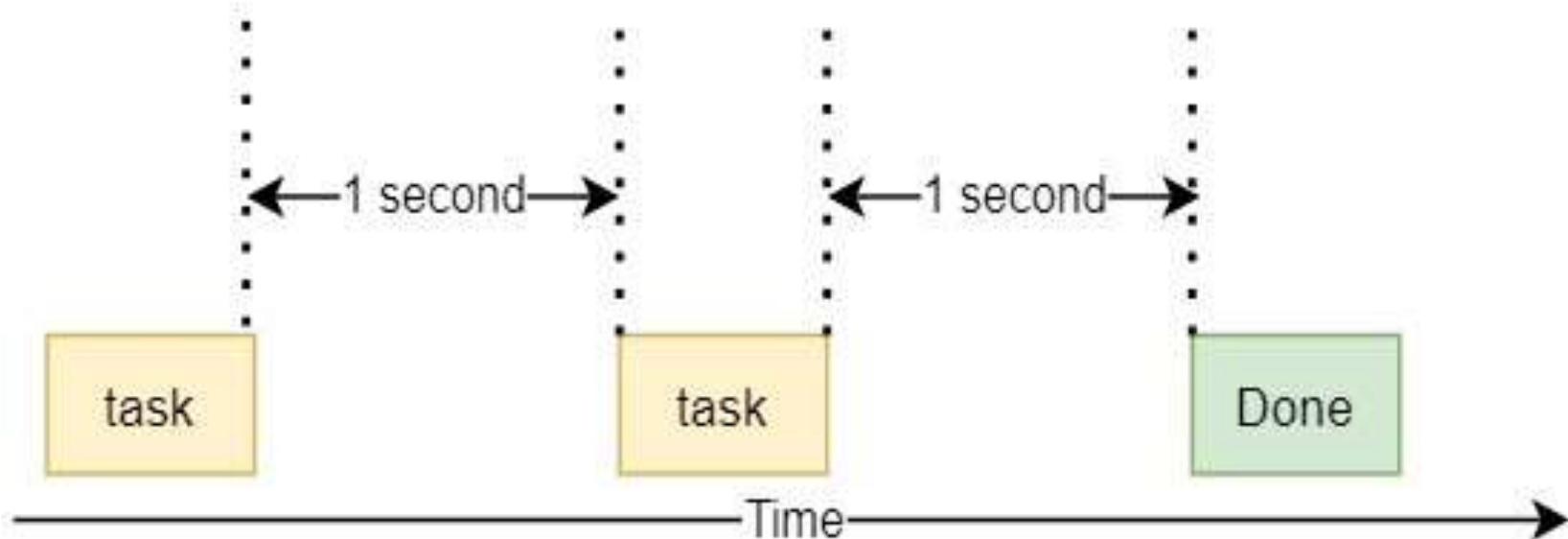
Output:

```
Starting a task...
done
Starting a task...
done
It took 2.01 second(s) to complete.
```

Threading



Flow diagram of previous Example:



Threading



By using thread:

Example:

```
from time import sleep, perf_counter
from threading import Thread

def task():
    print('Starting a task...')
    sleep(1)
    print('done')

start_time = perf_counter()
# create two new threads
t1 = Thread(target=task)
t2 = Thread(target=task)
# start the threads
t1.start()
t2.start()
# wait for the threads to complete
t1.join()
t2.join()
end_time = perf_counter()
print(f'It took {end_time - start_time: 0.2f} second(s) to complete.')
```

Output:

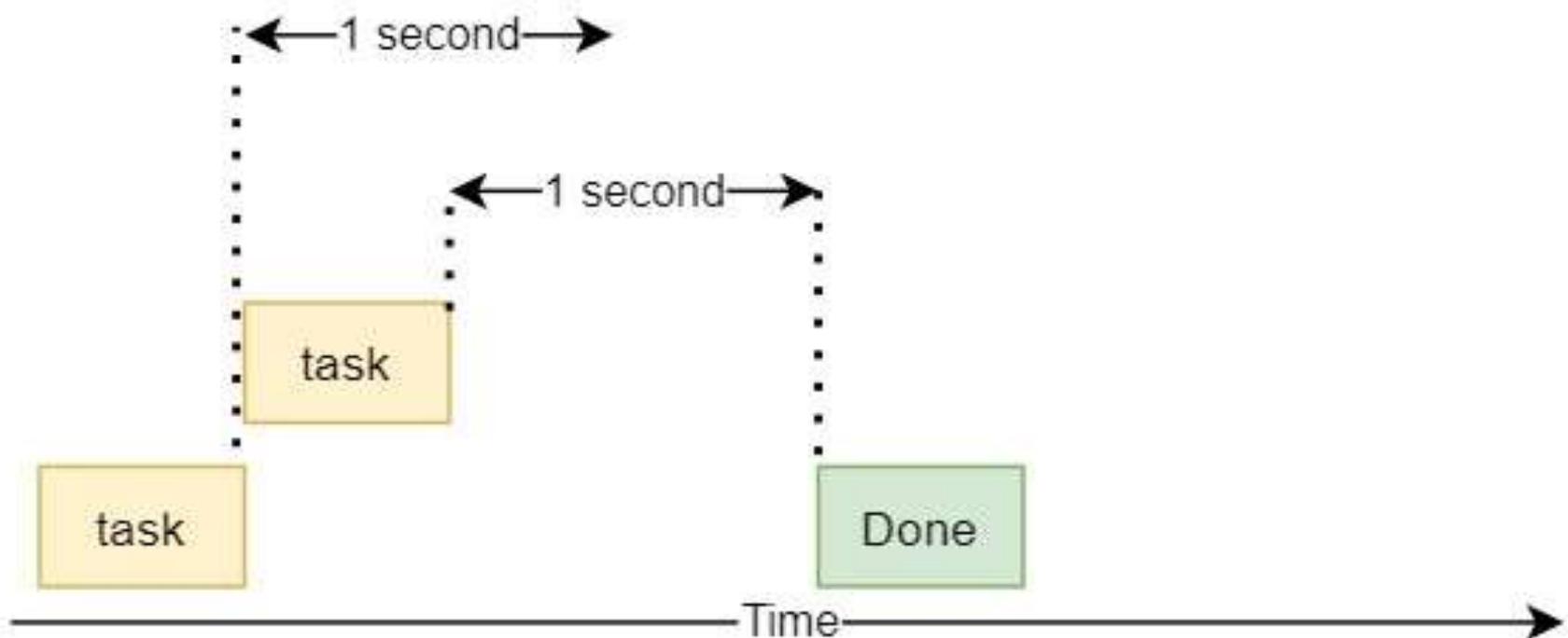
```
Starting a task...
Starting a task...
done
done

It took 1.02 second(s) to complete.
```

Threading



Flow diagram of previous Example:



Threading



Passing Arguments to threads:

Example:

```
from time import sleep, perf_counter
from threading import Thread
def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    print(f'The task {id} completed')
start_time = perf_counter()
# create and start 10 threads
threads = []
for n in range(1, 11):
    t = Thread(target=task, args=(n,))
    threads.append(t)
    t.start()
# wait for the threads to complete
for t in threads:
    t.join()
end_time = perf_counter()
print(f'It took {end_time - start_time: 0.2f} second(s) to complete.')
```

Output:

```
Starting the task 1...
Starting the task 2...
Starting the task 3...
Starting the task 4...
Starting the task 5...
Starting the task 6...
Starting the task 7...
Starting the task 8...
Starting the task 9...
Starting the task 10...
The task 10 completed
The task 8 completed
The task 1 completed
The task 6 completed
The task 7 completed
The task 9 completed
The task 3 completed
The task 4 completed
The task 2 completed
The task 5 completed
It took 1.02 second(s) to complete.
```

Threading



When to use Python Threading:

- *I/O bound tasks:* The I/O bound tasks spend more time doing I/O than doing computations. For Example: network requests, database connections and file reading/writing.
- *CPU-bound tasks:* The CPU bound tasks use more time doing computation than generating I/O requests. For Example: finding prime numbers, video streaming.

Threading



Difference between processes and threads:

Criteria	Process	Thread
Memory Sharing	Memory is not shared between processes	Memory is shared between threads within a process
Memory footprint	Large	Small
CPU-bound & I/O-bound processing	Optimized for CPU-bound tasks	Optimized for I/O bound tasks
Starting time	Slower than a thread	Faster than a process
Interruptability	Child processes are interruptible.	Threads are not interruptible.

Thank you



Collections

Collections



Collection:

- The collection module in python provides different types of containers.
- A Container is an object that is used to store different objects and provide a way to access the contained objects and iterate over them.
- Some of the built-in containers are tuple, list, Dictionary etc.

Collections



Types of collections:

- Counters
- OrderedDict
- DefaultDict
- ChainSet
- NamedTuple
- DeQue
- UserDict
- UserList
- UserString

Collections



Counters:

- A Counter is a sub-class of the dictionary.
- It is used to keep the count of the element in an iterable in the form of an unordered dictionary where the key represents the element in the iterable and value represents the count of that element in the iterable.

Example:

```
# A Python program to show different
# ways to create Counter
from collections import Counter

# With sequence of items
print(Counter(['B','B','A','B','C','A','B',
               'B','A','C']))

# with dictionary
print(Counter({'A':3, 'B':5, 'C':2}))

# with keyword arguments
print(Counter(A=3, B=5, C=2))
```

Output:

```
Counter({'B': 5, 'A': 3, 'C': 2})
Counter({'B': 5, 'A': 3, 'C': 2})
Counter({'B': 5, 'A': 3, 'C': 2})
```

Collections



OrderedDict:

- An OrderedDict is also a sub-class of dictionary but unlike dictionary, it remembers the order in which the keys were inserted.

Example:

```
# A Python program to demonstrate working
# of OrderedDict
from collections import OrderedDict
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4
print('Before Deleting')
for key, value in od.items():
|   print(key, value)
# deleting element
od.pop('a')
# Re-inserting the same
od['a'] = 1
print('\nAfter re-inserting')
for key, value in od.items():
|   print(key, value)
```

Output:

```
Before Deleting
a 1
b 2
c 3
d 4

After re-inserting
b 2
c 3
d 4
a 1
```

Collections



DefaultDict:

- A DefaultDict is also a sub-class to dictionary.
- It is used to provide some default values for the key that does not exist and never raises a KeyError.

Example:

```
# Python program to demonstrate
# defaultdict
from collections import defaultdict
# Defining the dict
d = defaultdict(int)
L = [1, 2, 3, 4, 2, 4, 1, 2]
# Iterate through the list
# for keeping the count
for i in L:
    # The default value is 0
    # so there is no need to
    # enter the key first
    d[i] += 1
print(d)
```

Output:

```
defaultdict(<class 'int'>, {1: 2, 2: 3, 3: 1, 4: 2})
```

Collections



ChainMap:

- A ChainMap encapsulates many dictionaries into a single unit and returns a list of dictionaries.

Example:

```
# Python program to demonstrate
# ChainMap
from collections import ChainMap

d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}
d3 = {'e': 5, 'f': 6}
# Defining the chainmap
c = ChainMap(d1, d2, d3)
print(c)
```

Output:

```
ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6})
```

Collections



NamedTuple:

- A NamedTuple returns a tuple object with names for each position which the ordinary tuples lack.
- For Example: consider a tuple named student where the first element represents fname, second name represents lname and the third element represents the DOB.
- Suppose for calling fname instead of remembering the index position you can actually call the element by using the fname argument.

Syntax:

```
namedtuple(typename, field_names)
```

Collections



Example:

```
# Python code to demonstrate namedtuple()
from collections import namedtuple

# Declaring namedtuple()
Student = namedtuple('Student',['name','age','DOB'])

# Adding values
S = Student('Nandini','19','2541997')

# Access using index
print ("The Student age using index is : ",end="")
print (S[1])

# Access using name
print ("The Student name using keyname is : ",end="")
print (S.name)
```

Output:

```
The Student age using index is : 19
The Student name using keyname is : Nandini
```

Collections



NamedTuple:

Conversion Operations:

- `_make()`: This function is used to return a namedtuple() from the iterable passed as argument.

- `_asdict()`: This function returns the OrderedDict() as constructed from the mapped values of namedtuple().

Collections



Example:

```
# Python code to demonstrate namedtuple() and
# _make(), _asdict()
from collections import namedtuple
# Declaring namedtuple()
Student = namedtuple('Student', ['name', 'age', 'DOB'])
# Adding values
S = Student('Nandini', '19', '2541997')

# initializing iterable
li = ['Manjeet', '19', '411997']

# initializing dict
di = { 'name' : "Nikhil", 'age' : 19 , 'DOB' : '1391997' }

# using _make() to return namedtuple()
print ("The namedtuple instance using iterable is : ")
print (Student._make(li))

# using _asdict() to return an OrderedDict()
print ("The OrderedDict instance using namedtuple is : ")
print (S._asdict())
```

Output:

```
The namedtuple instance using iterable is :
Student(name='Manjeet', age='19', DOB='411997')
The OrderedDict instance using namedtuple is :
{'name': 'Nandini', 'age': '19', 'DOB': '2541997'}
```

Collections



Deque:

- Deque(Doubly Ended Queue) is the optimized list for quicker append and pop operations from both sides of the container.
- It provides O(1) time complexity for append and pop operations as compared to list with O(n) time complexity.

Example:

```
# Python code to demonstrate deque
from collections import deque
# Declaring deque
queue = deque(['name', 'age', 'DOB'])
print(queue)
```

Output:

```
deque(['name', 'age', 'DOB'])
```

Collections



Inserting elements in Deque:

- Elements in deque can be inserted from both ends.
- To insert element from right append() method is used and to insert the elements from the left appendLeft() method is used.

Example:

```
# Python code to demonstrate working of
# append(), appendleft()
from collections import deque
# initializing deque
de = deque([1,2,3])
# using append() to insert element at right end
# inserts 4 at the end of deque
de.append(4)
# printing modified deque
print ("The deque after appending at right is : ")
print (de)
# using appendleft() to insert element at left end
# inserts 6 at the beginning of deque
de.appendleft(6)
# printing modified deque
print ("The deque after appending at left is : ")
print (de)
```

Output:

```
The deque after appending at right is :
deque([1, 2, 3, 4])
The deque after appending at left is :
deque([6, 1, 2, 3, 4])
```

Collections



Removing elements in Deque:

- Elements can also be removed from the deque from both the ends.
- To remove elements from right use pop() method and to remove elements from the left use popleft() method.

Example:

```
# Python code to demonstrate working of
# pop(), and popleft()

from collections import deque

# initializing deque
de = deque([6, 1, 2, 3, 4])

# using pop() to delete element from right end
# deletes 4 from the right end of deque
de.pop()

# printing modified deque
print ("The deque after deleting from right is : ")
print (de)

# using popleft() to delete element from left end
# deletes 6 from the left end of deque
de.popleft()

# printing modified deque
print ("The deque after deleting from left is : ")
print (de)
```

Output:

```
The deque after deleting from right is :
deque([6, 1, 2, 3])
The deque after deleting from left is :
deque([1, 2, 3])
```

Collections



UserDict:

- UserDict is a dictionary-like container that acts as a wrapper around the dictionary objects.
- This container is used when someone wants to create their own dictionary with some modified or new functionality.

Syntax:

```
.UserDict([initialdata])
```

Collections



Example:

```
# Python program to demonstrate
# userdict
from collections import UserDict
# Creating a Dictionary where
# deletion is not allowed
class MyDict(UserDict):
    # Function to stop deletion
    # from dictionary
    def __del__(self):
        raise RuntimeError("Deletion not allowed")
    # Function to stop pop from
    # dictionary
    def pop(self, s = None):
        raise RuntimeError("Deletion not allowed")
    # Function to stop popitem
    # from Dictionary
    def popitem(self, s = None):
        raise RuntimeError("Deletion not allowed")
# Driver's code
d = MyDict({'a':1,
            'b': 2,
            'c': 3})
d.pop(1)
```

Output:

```
Exception has occurred: RuntimeError X
Deletion not allowed

File "D:\Syllabus\Python\Full Syllabus\Core Python\Gama\Practice\collections.py", line 15, in pop
    raise RuntimeError("Deletion not allowed")
File "D:\Syllabus\Python\Full Syllabus\Core Python\Gama\Practice\collections.py", line 24, in <module>
    d.pop(1)
RuntimeError: Deletion not allowed
```

Collections



UserList:

- UserList is a list like container that acts as a wrapper around the list objects.
- This is useful when someone wants to create their own list with some modified or additional functionality.

Syntax:

```
UserList([list])
```

Collections



Example:

```
# Python program to demonstrate
# userlist
from collections import UserList
# Creating a List where
# deletion is not allowed
class MyList(UserList):
    # Function to stop deletion
    # from List
    def remove(self, s = None):
        raise RuntimeError("Deletion not allowed")
    # Function to stop pop from
    # List
    def pop(self, s = None):
        raise RuntimeError("Deletion not allowed")
# Driver's code
L = MyList([1, 2, 3, 4])
print("Original List")
# Inserting to List"
L.append(5)
print("After Insertion")
print(L)
# Deleting From List
L.remove()
```

Output:

Original List
After Insertion
[1, 2, 3, 4, 5]

```
Exception has occurred: RuntimeError
Deletion not allowed
File "D:\Syllabus\Python\Full Syllabus\Core Python\Gana\Practice\collections.py", line 11, in remove
    raise RuntimeError("Deletion not allowed")
File "D:\Syllabus\Python\Full Syllabus\Core Python\Gana\Practice\collections.py", line 24, in <module>
    L.remove()
RuntimeError: Deletion not allowed
```

Collections



UserString:

- UserString is a string like container that acts as a wrapper around the string objects.
- It is used when someone wants to create their own strings with some modified or additional functionality.

Syntax:

```
UserString(seq)
```

Collections



Example:

```
# Python program to demonstrate
# userstring
from collections import UserString
# Creating a Mutable String
class Mystring(UserString):
    # Function to append to
    # string
    def append(self, s):
        self.data += s

    # Function to remove from
    # string
    def remove(self, s):
        self.data = self.data.replace(s, "")

# Driver's code
s1 = Mystring("Hello User")
print("Original String:", s1.data)
# Appending to string
s1.append("s")
print("String After Appending:", s1.data)
# Removing from string
s1.remove("e")
print("String after Removing:", s1.data)
```

Output:

```
Original String: Hello User
String After Appending: Hello Users
String after Removing: Hllo Usrs
```

Thank You