

FREE NOTES FOR BEGINNER



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 1-INTRODUCTION



Today's Agenda

- Prerequisites Of Learning Python
- Necessity Of Programming
- What Is Python ?
- Why And Who Created It ?
- What Python Can Do ?
- Why Should I Learn Python In 2020 ?
- Important Features
- Course Outline

What You Should Know ?



To start learning **Python**, there is no strict **pre-requisite**

No specific programming language knowledge is needed.

Just **basic knowledge** in **C/C++** is more than **sufficient**



Why Do We Need Programming ?

- To **communicate** with **digital machines** and make them work accordingly
- Today in the **programming world**, we have more than **900 programming languages** available.
- And **every language** is designed to fulfill a **particular kind of requirement**



Brief History Of Prog. Lang

C language was primarily designed to develop “**System Softwares**” like **Operating Systems**, **Device Drivers** etc .

To remove **security problems** with “C” language , **C++ language** was designed.

It is an **Object Oriented Language** which provides **data security** and can be used to solve **real world problems**.

Many **popular softwares** like **Adobe Acrobat** , **Winamp Media Player**,**Internet Explorer**,**MS Office** etc were designed in **C++**

Courtsey:<http://www.stroustrup.com/applications.html>



What is **Python** ?

Python is a **general purpose** and **powerful** programming language.

Python is considered as one of the **most versatile programming language** as it can be used to develop almost **any kind of application** including :

- **desktop applications**
- **web applications**
- **IoT applications**
- **AI, ML and Data Science applications**
- **and many more . . .**



Who created **Python**?



Developed by **Guido van Rossum**, a **Dutch** scientist

Created at **Center For Mathematics and Research**, **Netherland**

It is **inspired** by another **programming language** called **ABC**



Why was **Python** created ?

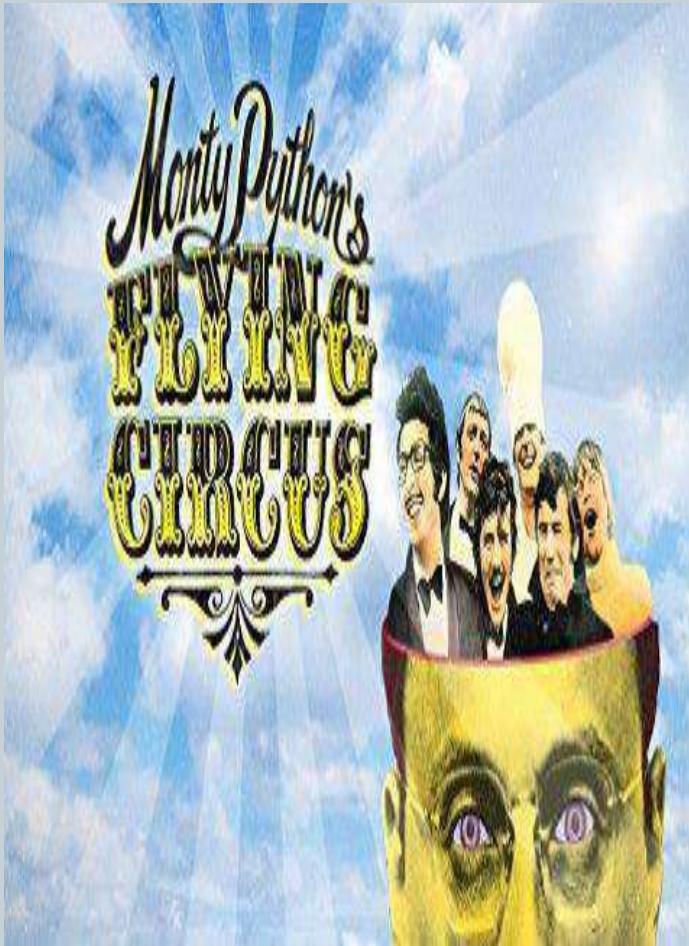


Guido started **Python** development as a **hobby** in **1989**

But since then it has grown to become **one of the most polished languages** of the computing world.



How Python got its name?



The name **Python** is inspired from **Guido's** favorite **Comedy TV show** called "**Monty Python's Flying Circus**"

Guido wanted **a name** that was **short**, **unique**, and **slightly mysterious**, so he decided to call the language **Python**.



Who manages **Python** today ?



From **version 2.1** onwards ,
Python is managed by **Python Software Foundation** situated
in **Delaware , USA**

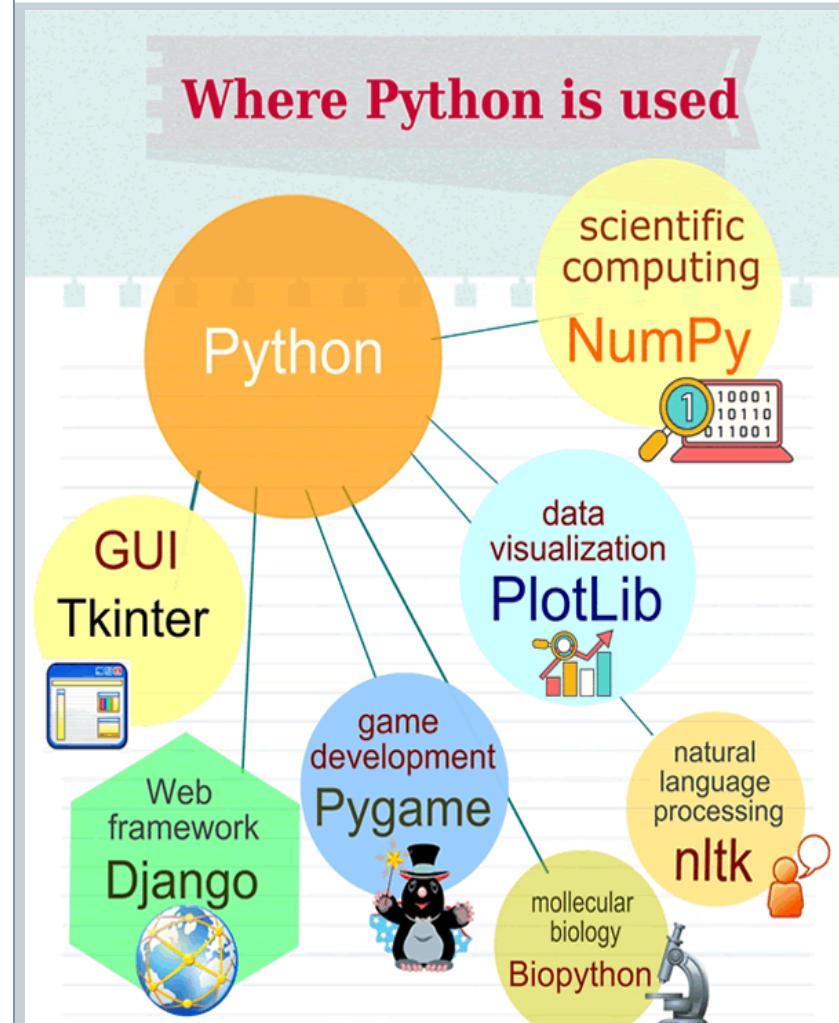


It is **a non-profit organization**
devoted to the growth and
enhancement of **Python**
language

Their website is
<http://www.python.org>



Where Is Python used?



GUI

Web

Data Science

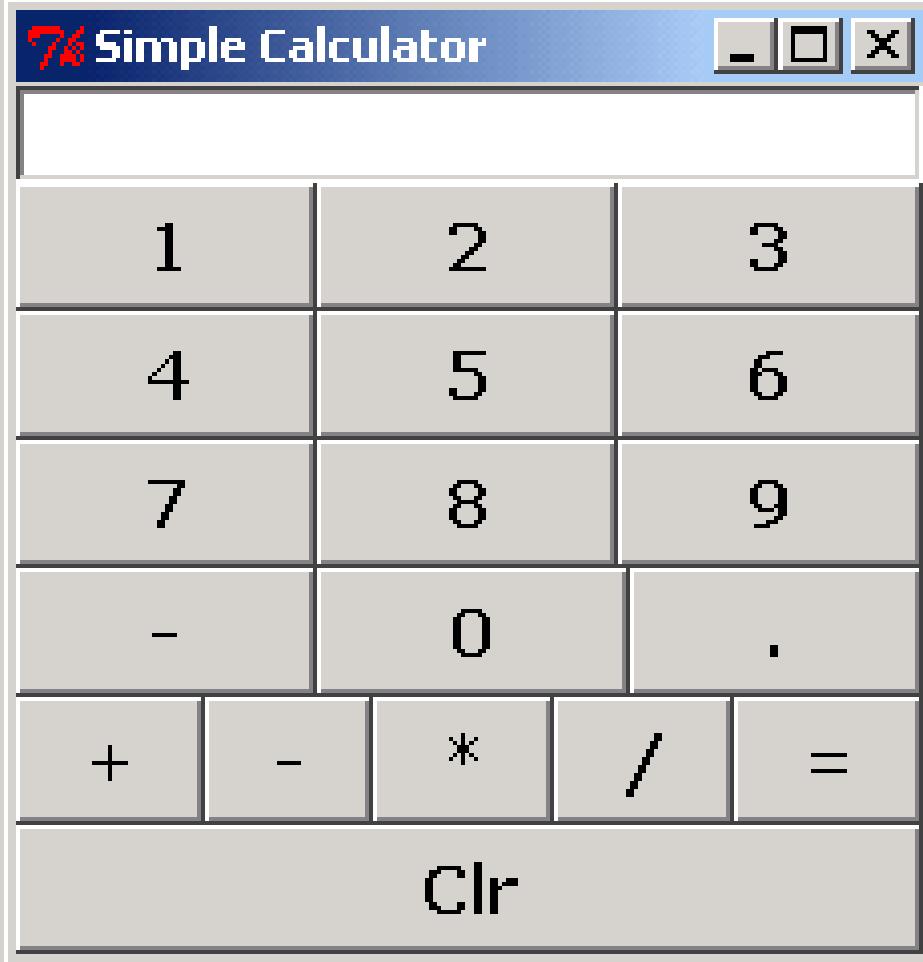
AI & ML

IoT

Hacking



GUI In Python

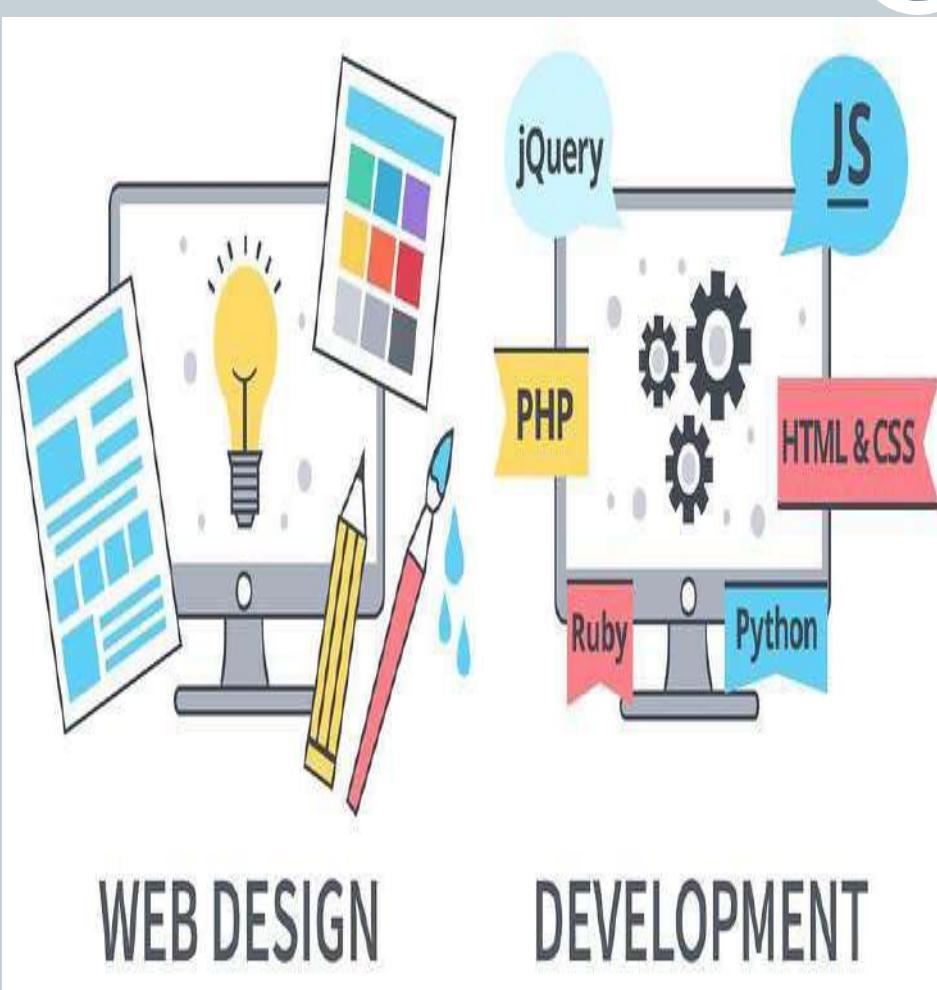


Python is used for GUI apps all the time.

It has famous libraries like PyQt , Tkinter to build desktop apps.



Web Application In **Python**



We can use **Python** to create **web applications** on many levels of complexity



Famous Websites Developed Using Pytho

There are **numerous examples** of **popular, high-load websites/webapps** that have been developed using **Python**.

Here are some of the **most popular** of them:

- NASA
- Instagram
- Udemy
- Spotify
- Mozilla
- Dropbox

And above all **YouTube**



Web Application In **Python**



There are many excellent **Python frameworks** like **Django**, **Flask** for **web application development**



Data Science In Python



Data Science is about
making **predictions** with
data

Some Examples



How do you think Super Market stores decide what are the items they should club together to make a combo?

How it happens ?

Answer: *Data Science*



Some Examples



Have you noticed that every time you log on to Google, Facebook and see ads, they are based on your preferences

How it happens ?

Answer: **Data Science**



Data Science In Python



Python is the **leading language** of choice for many **data scientists**

It has **grown in popularity** due to it's excellent **libraries** like **Numpy , Pandas** etc



AI&MLIn Python

Machine learning is a field of **AI(Artificial Intelligence)** by using which **software applications** can predict more accurate outcomes based on historical data.



A graphic featuring a blue circular icon containing a white lightbulb, symbolizing ideas or intelligence. To the right of the icon is the Python logo. Below these elements, the words "MACHINE" and "LEARNING" are written in large, bold, black and red letters respectively, with a slight gradient effect.

MACHINE**LEARNING**

It is heavily used in **Face recognition** , **music recommendation** , **medical data** etc



Use Of ML In COVID-19



news-medical.net/news/20200520/Scientists-use-machine-learning-methods-to-estimate-COVID-19s-seasonal-cycle.aspx



MEDICAL HOME

LIFE SCIENCES HOME

Become a Member

Search...



About Coronavirus News Health A-Z Drugs Medical Devices Interviews White Papers More...

Scientists use machine learning methods to estimate COVID-19's seasonal cycle

Download PDF Copy



1



in



f



t



r



Reviewed by Emily Henderson, B.Sc.

May 20 2020

One of the many unanswered scientific questions about COVID-19 is whether it is seasonal like the flu - waning in warm summer months then resurging in the fall and winter. Now scientists at Lawrence Berkeley National Laboratory (Berkeley Lab) are launching a project to apply machine-learning methods to a plethora of health and environmental datasets, combined with high-resolution climate models and seasonal forecasts, to tease out the answer.

“Environmental variables, such as temperature, humidity, and UV [ultraviolet radiation] exposure, can have an effect on the virus directly, in terms of its viability. They can also affect the transmission of the virus and the formation of

ADVERTISEMENT

Trending Stories

Latest Interviews

Top Health Articles



Using Qigong to manage COVID-19 in older adults

SARS-CoV-2 is uniquely



AI&MLIn **Python**

Python has many wonderful **libraries** to implement ML algos like **SciKit-Learn , Tensorflow** etc





IoT In Python



The **Internet of Things**, or **IoT**, refers to the **billions of physical devices** around the world that are now **connected to the internet**, all **collecting** and **sharing** data.

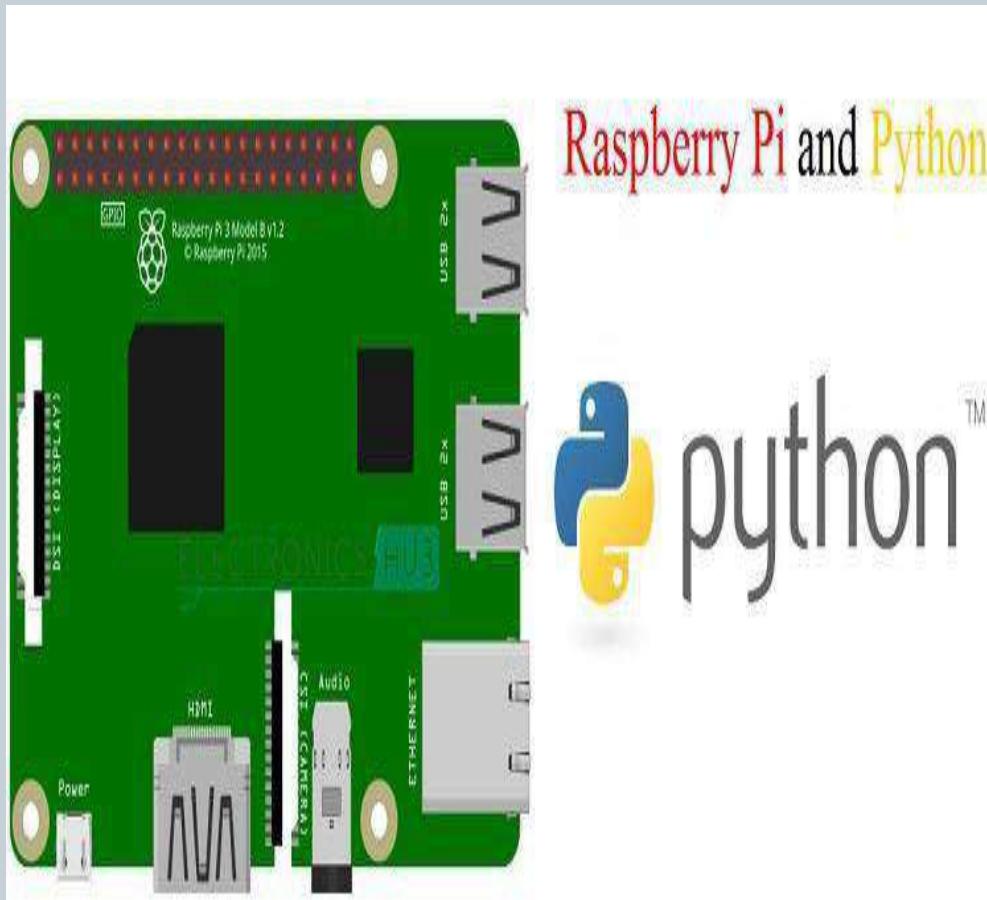
For example:

- A **lightbulb** that can be **switched on** using a **smartphone app** is an **IoT device**



IoT In Python

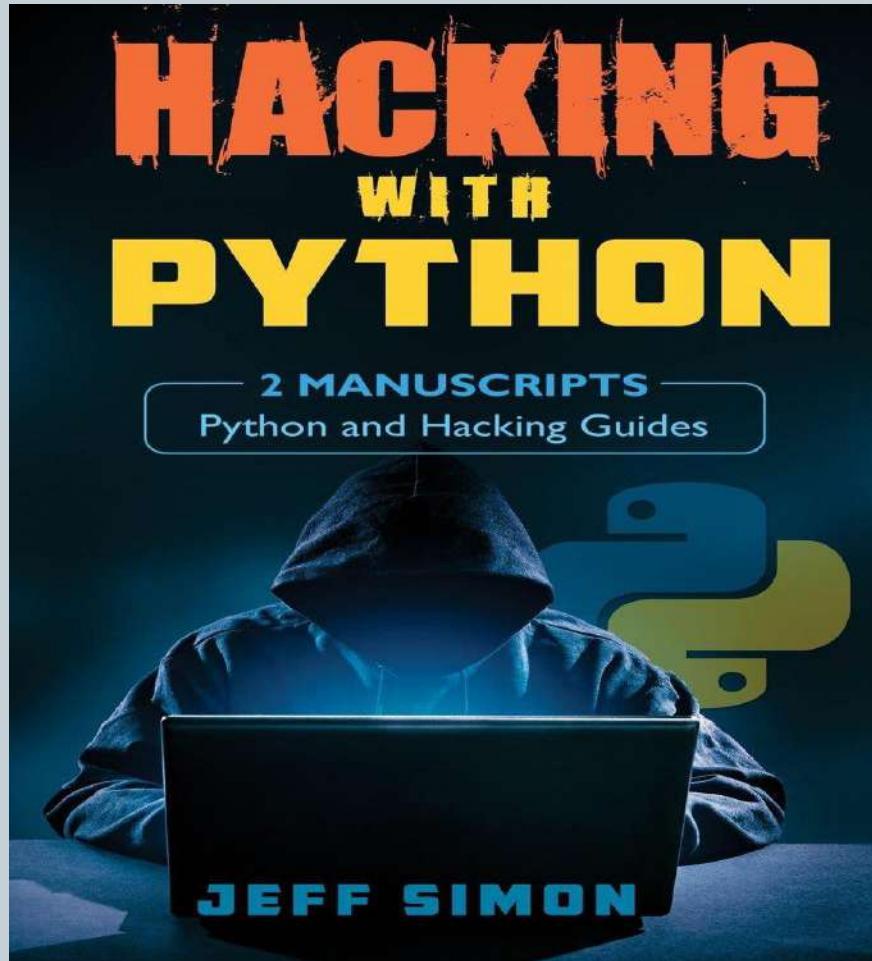
We can build **Home Automation System** and even **robots** using **IoT**



The **coding** on an **IoT platform** like **Raspberry Pi** can be **performed** using **Python**



Hacking In Python



Python has gained popularity as preferred language for hacking.

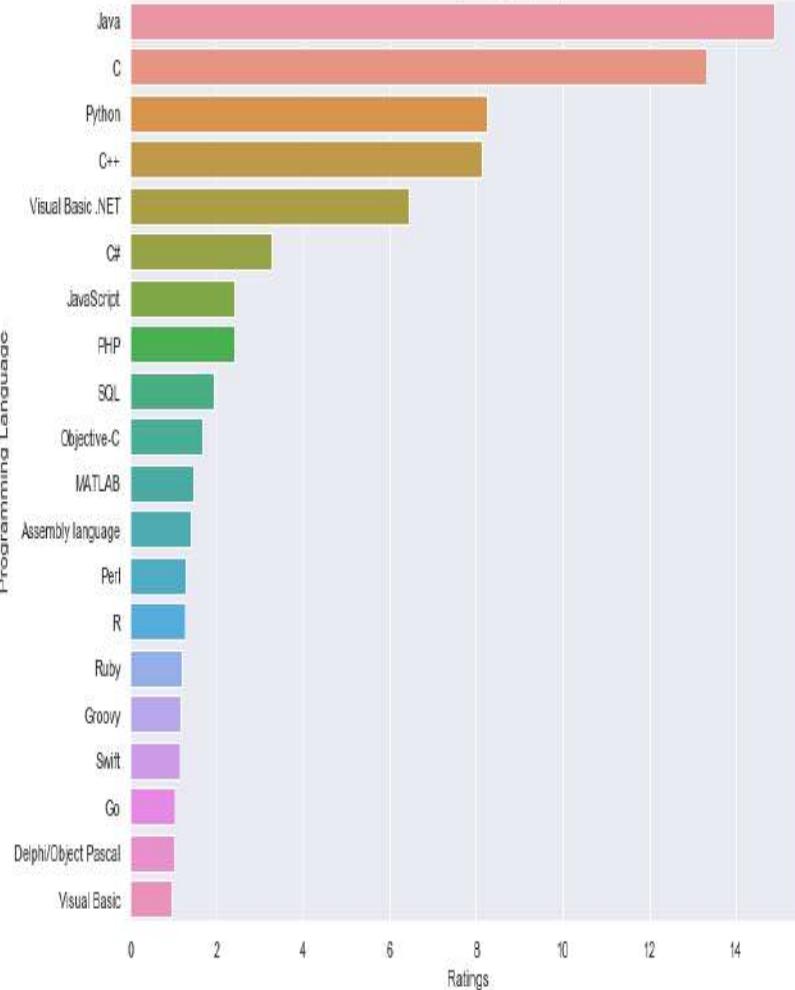
Hackers generally develop small scripts and Python provides amazing performance for small programs



Why should I learn Python ?



Mar 2019 - Programming Popularity



3rd most popular programming

Fastest growing language

Opens lots of doors

Big corporates prefer Python

Means , **PYTHON IS THE FUTURE**



Who uses **Python** today ?



Who all are using Python?



YAHOO!

Google

You **Tube**



reddit

BitTorrent

IBM



Dropbox



redhat

CANONICAL

NETFLIX

Quora



and the list goes on...



Features Of Python



Simple

Dynamically Typed

Robust

Supports multiple programming paradigms

Compiled as well as Interpreted

Cross Platform

Extensible

Huge Library



Simple



Python is very simple

As compared to other popular languages like **Java** and **C++**, it is easier to code in **Python**.

Python code is comparatively 3 to 5 times smaller than **C/C++/Java** code



Print Hello Bhopal!

INC

```
#include <stdio.h>
int main(){
    printf("Hello Bhopal!");
    return 0;
}
```

IN JAVA

```
public class HelloWorld{
    public static void main(String[] args){
        System.out.println( "Hello Bhopal!");
    }
}
```

IN PYTHON

```
print('Hello Bhopal!')
```



These Notes Have Python_World_In Notes Copyrights under copy
right
Acts of Government of India. So commercial Use of this notes is
Strictly prohibited



Add 2 Nos



IN C

```
#include <stdio.h>
int main(){
int a=10,b=20;
printf("Sum is%d",a+b);
return 0;
}
```

IN JAVA

```
public class HelloWorld{
    public static void main( String[] args ) {
int a=10,b=20;
System.out.println( "Sum is "+(a+b));
    }
}
```

IN PYTHON

```
a,b=10,20
print("Sum is",a+b)
```



Swap 2 Nos



IN C

....

```
temp=a;  
a=b;  
b=temp;
```

IN JAVA

...

```
temp=a;  
a=b;  
b=temp;
```

IN PYTHON

...

```
a,b=b,a
```



Dynamically Typed

Dynamically typed vs Statically typed

Statically Typed (C/C++/Java)

- Need to declare variable type before using it
- Cannot change variable type at runtime
- Variable can hold only one type of value throughout its lifetime

Dynamically Typed – Python

- Do not need to declare variable type
- Can change variable type at runtime
- Variable can hold different types of value throughout its lifetime



Dynamically Typed



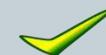
IN C

```
int a;  
a=10;  
a="Bhopal";
```



IN Python

```
a=10  
a="Bhopal"
```





Robust



Python has very **strict rules** which every program must **compulsorily follow** and if these rules are **violated** then Python terminates the code by generating "**Exception**"

To understand **Python's** robustness , guess the output of the following **C/C++** code:

```
int arr[5];
int i;
for(i=0;i<=9;i++)
{
arr[i]=i+1;
}
```



Robust



In **Python** if we write the same code then it will generate **Exception** terminating the code

Due to this **other running programs** on the computer **do not get affected** and the system remains **safe** and **secure**

Supports Multiple Programming Paradigms



Python supports both **procedure-oriented** and **object-oriented** programming which is one of the key features.

In ***procedure-oriented*** languages, the program is built around **procedures** or **functions** which are nothing but reusable pieces of programs.

In ***object-oriented*** languages, the program is built around **objects** which combine **data** and **functionality**

Compiled As Well As Interpreted



Python uses both a **compiler** as well as **interpreter** for converting our source and running it

However , the **compilation part** is **hidden** from the programmer ,so mostly people say it is an **interpreted language**



Cross Platform



Let's assume we've written a **Python** code for our **Windows machine**.

Now, if we want to run it on a **Mac**, we **don't need to make changes** to it for the same.

In other words, we can take one code and run it on any machine, **there is no need to write different code for different machines.**

This makes **Python** a **cross platform language**



Extensible



Python allows us to call **C/C++/Java** code from a **Python** code and thus we say it is an **extensible language**

We generally use this **feature** when we need a **critical piece of code** to run **very fast**.

So we can code that part of our program in **C** or **C++** and then use it from our **Python** program.



Huge Library

The Python Standard Library is **huge** indeed.

Click to add text

It can help you do various things like **Database Programming**,
E-mailing, **GUI Programming** etc

These Notes Have Python_world In Note copy rights under copy right
Acts of Government of India. So commercial usages of this notes is Strictly prohibited

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



PYTHON

LECTURE 2

Today's Agenda



Version History, Downloading and Installing Python

- **Version History**
- **Python 2 v/s Python 3**
- **Different Python Implementations**
- **Downloading And Installing Python**
- **Testing Python Installation**

Python Version History



- First released on Feb-20th -1991 (ver 0.9.0)
- Python 1.0 launched in Jan-1994
- Python 2.0 launched in Oct-2000
- Python 3.0 launched in Dec-2008
- Python 2.7 launched in July 2010
- Python 3.6.5 launched on March-28th-2018
- Python 3.7 launched on June-27th -2018
- Python 3.8.0 lunched On Oct-14th -2019
- Current latest (as of now) is Python 3.8.5 launched on Jul-20th-2020

The Two Versions Of Python



- As you can observe from the previous slide , there are 2 major versions of **Python** , called **Python 2** and **Python 3**
- Python 3** came in **2008** and it is not **backward compatible** with **Python 2**
- This means that a project which uses **Python 2** will not run on **Python 3**.
- This means that we have to **rewrite the entire project** to migrate it from **Python 2** to **Python 3**



Some Important Differences

- In Python 2
`print "Hello Bhopal"`
- In Python 3
`print("Hello Bhopal")`
- In Python 2
 $5/2 \rightarrow 2$
 $5/2.0 \rightarrow 2.5$
- In Python 3
 $5/2 \rightarrow 2.5$
- The way of accepting input has also changed and like this there are many changes



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

The Two Versions Of Python

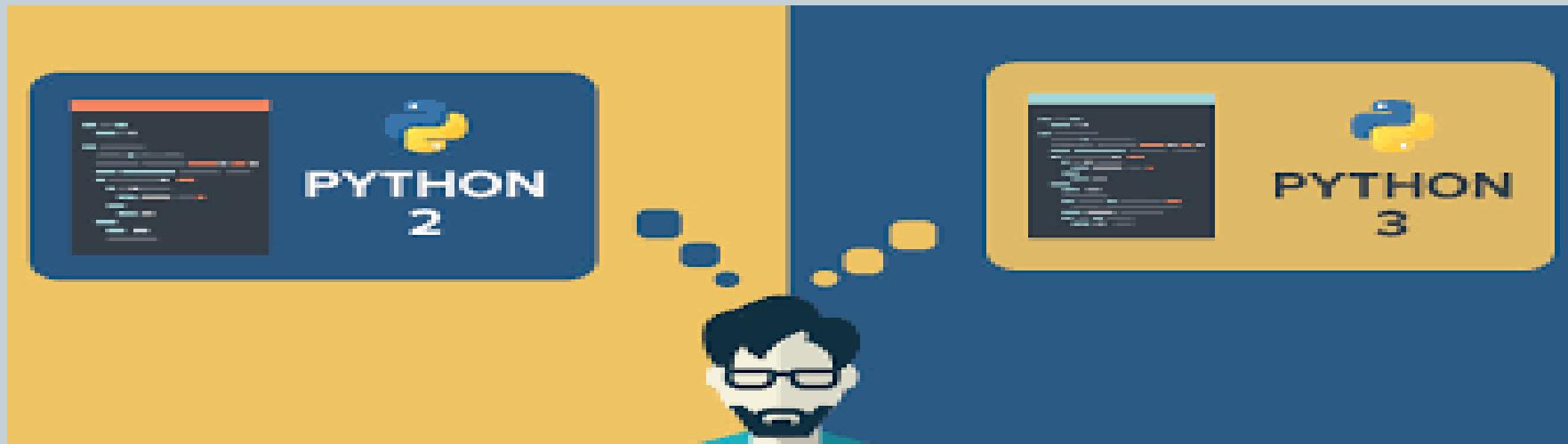


- So to prevent this overhead of programmers , **PSF** decided to support **Python 2** also.
- But this support was only till **Jan-1-2020**
- You can visit **<https://pythonclock.org/>** to see exactly how much time before **Python 2** has retired.

Which Version Should I Use ?



- For beginners , it is a point of confusion as to **which Python version they should learn ?**



- The obvious answer is **Python 3**



Why Python 3?



- We should go with **Python 3** as it brings lot of new features and new tricks compared to **Python 2**
- Moreover as per PSF, ***Python 2.x is legacy, Python 3.x is the present and future of the language***
- All major future upgrades will be to **Python 3** and , **Python 2.7** will never move ahead to even **Python 2.8**



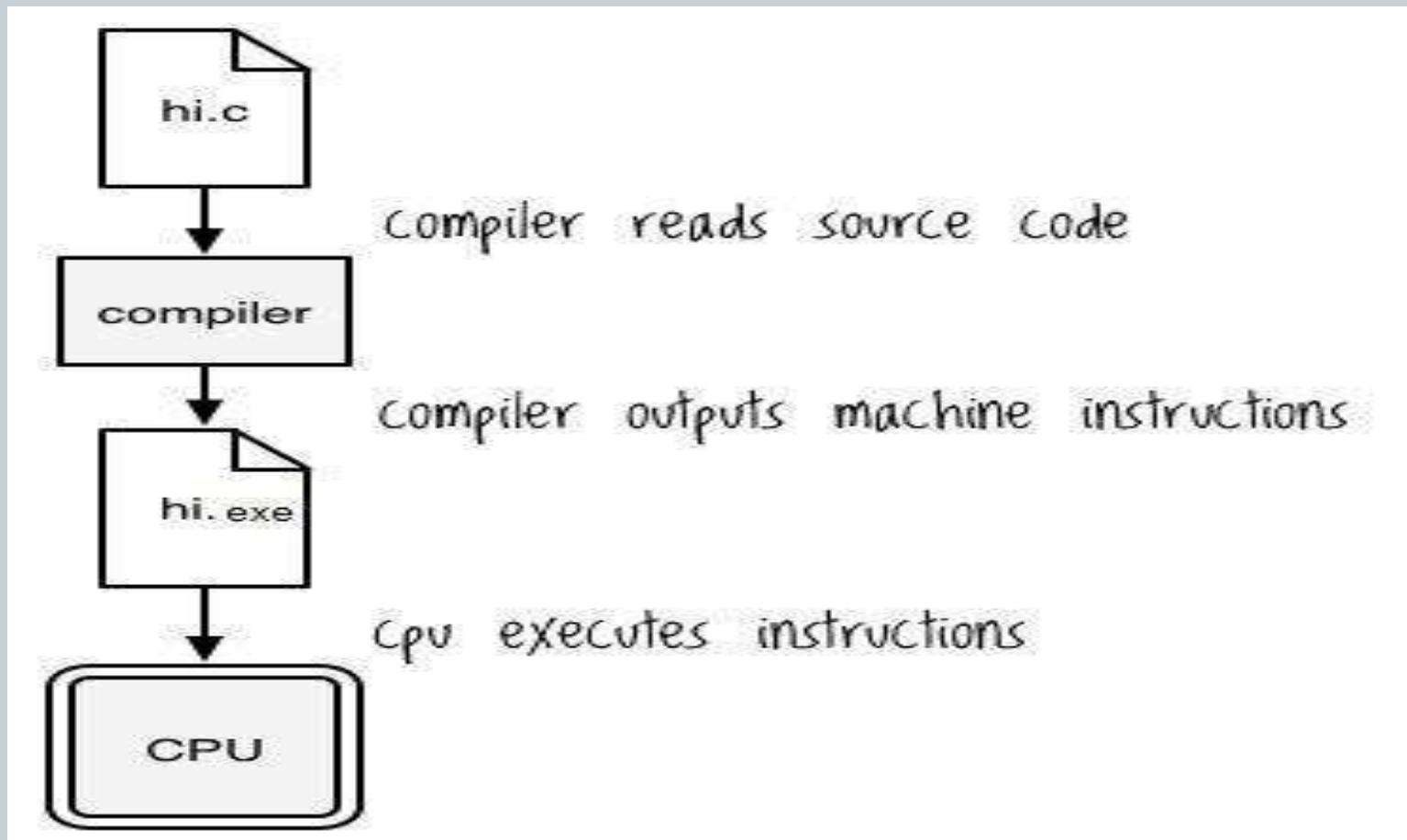
Various Implementations Of Python

The **Python language** has many popular **implementations**

The word **implementation** means **the tools/software** which are used for the execution of programs written in the **Python language**.

As of now **Python** has around **26 implementations**, but the most common are: **Cpython**, **Jython**, **IronPython** and **PyPy**

Difference Between Machine Code And ByteCode



Benefits And Drawbacks Of Machine Code

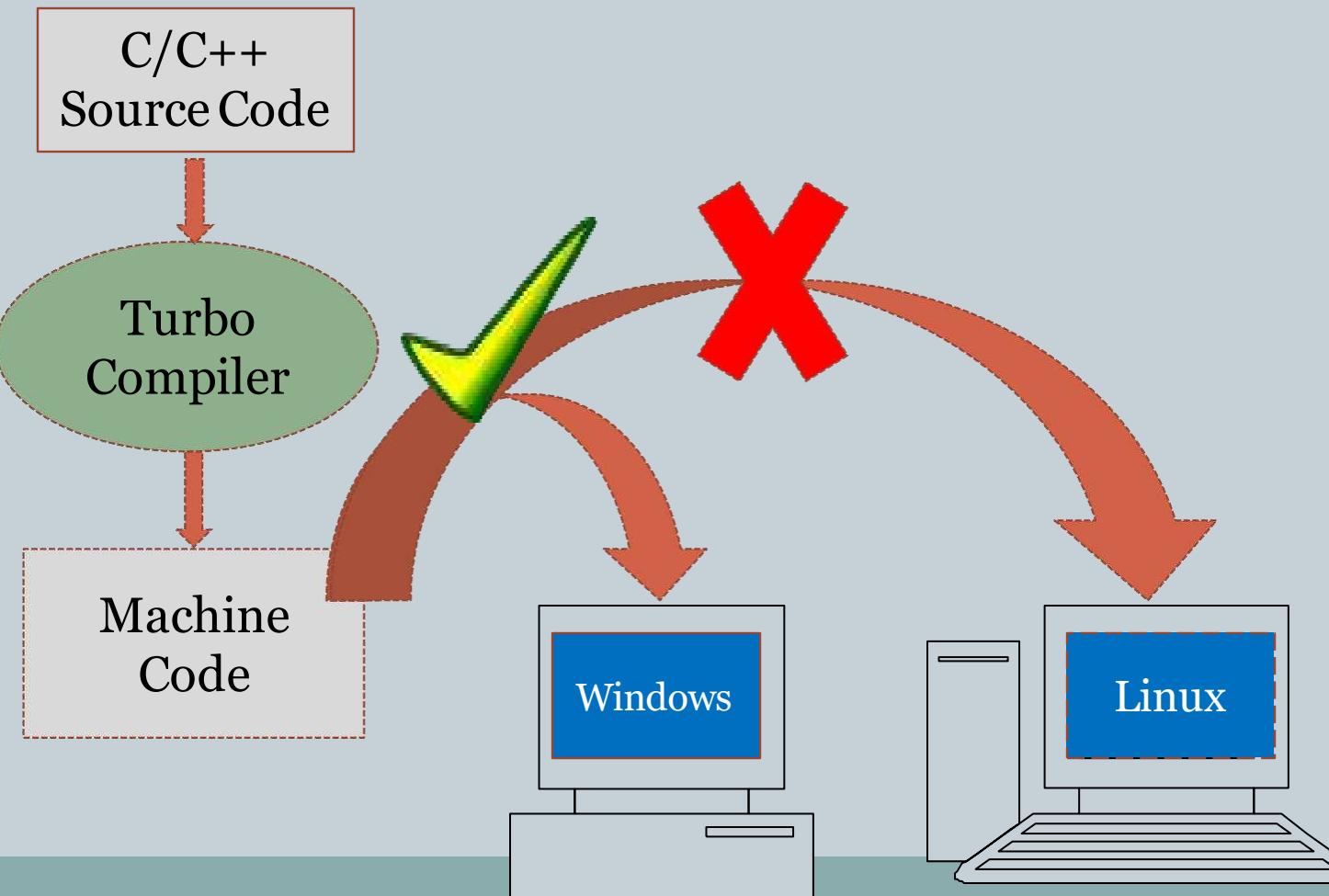


The benefit of **machine code** is that **it runs very fast** because it is in the **native form** i.e a form which is directly understandable to the **CPU**.

However the **drawback** is that it **cannot run** on **another platform** which is **different** than the **platform** on which the code was **compiled**.

In simple words , the **.exe** file of a C program compiled in Windows cannot run on Linux or Mac because every platform (OS+CPU) has it's own **machine code instruction set**.

Benefits And Drawbacks Of Machine Code



Difference Between Machine Code And ByteCode



Bytecode

Bytecode is an **intermediate code** but it is different than **machine code** because it **cannot be directly executed** by the **CPU**.

So whenever the **compiler** of a **language** which supports **bytecode** compiles a program , the **compiler** never generates **machine code**.

Rather it generates a **machine independent code** called the **“bytecode”**.

Difference Between Machine Code And ByteCode



Now since this **bytecode** is not **directly understandable** to the platform(OS &CPU) , so another special layer of software is **required** to convert these **bytecode** instructions to **machine dependent form** .

This special layer is called **VM** or **Virtual Machine**.

Difference Between Machine Code And ByteCode



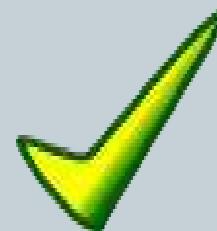
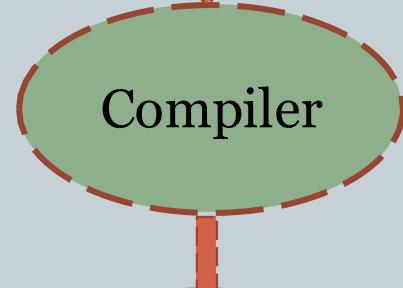
Python is one of the languages which works on the concept of **VM**.

Thus any such **platform** for which a **VM** (called **PVM** in **Python**) is available can be used to execute a **Python program** irrespective of where it has been **compiled**.

Program Execution in Python

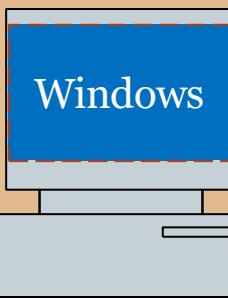


Source Code

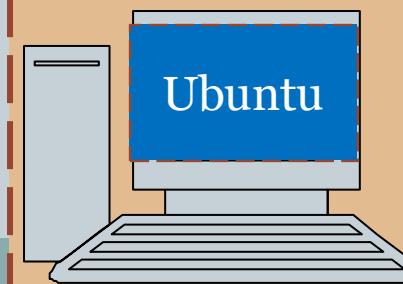


Byte Code

PVM



PVM



PVM



Benefits And Drawbacks Of ByteCode



The benefit of **bytecode** is that it makes our program **platform independent** i.e. we only have to write the program once and we can run it any platform provided there is a **VM** available on that platform

However the **drawback** is that **it runs at a slower pace** because the interpreter inside the **VM** has to **translate** each **bytecode** instruction to **native form** and then send it for execution to the **CPU**.



C_Python



The **default implementation** of the **Python** programming language is **C_Python** which is written in **Clanguage**.

C_Python is the original **Python** implementation and it is the implementation we will download from [**Python.org**](https://www.python.org).

People call it **C_Python** to distinguish it from other **Python** implementations

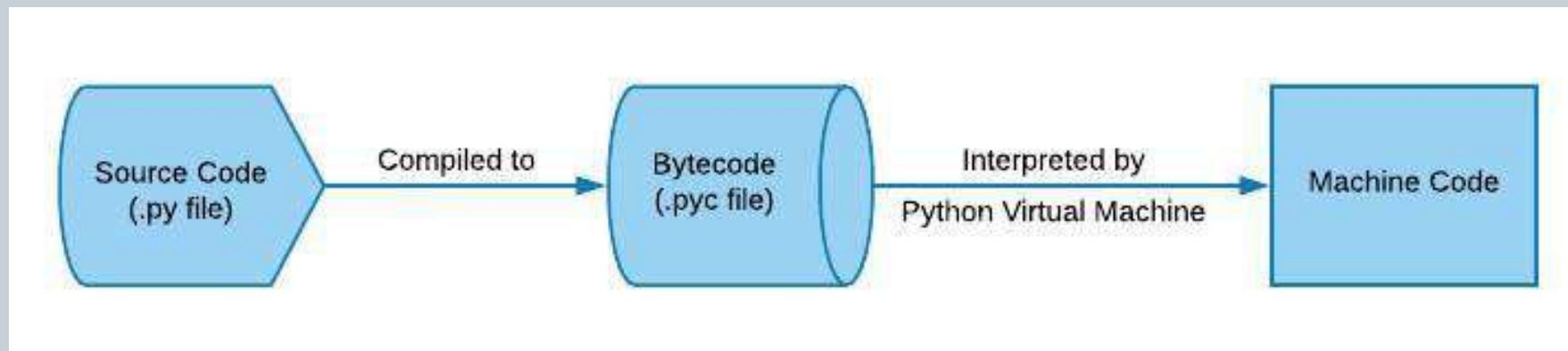
Also we must understand that **Python** is the language and **C_Python** is its **compiler/interpreter** written in **Clanguage** to run the **Python** code.



C_Python



C_Python compiles the **python source code** into intermediate **bytecode**, which is executed by the **C_Python virtual machine** also called as the **PVM**.





Jython



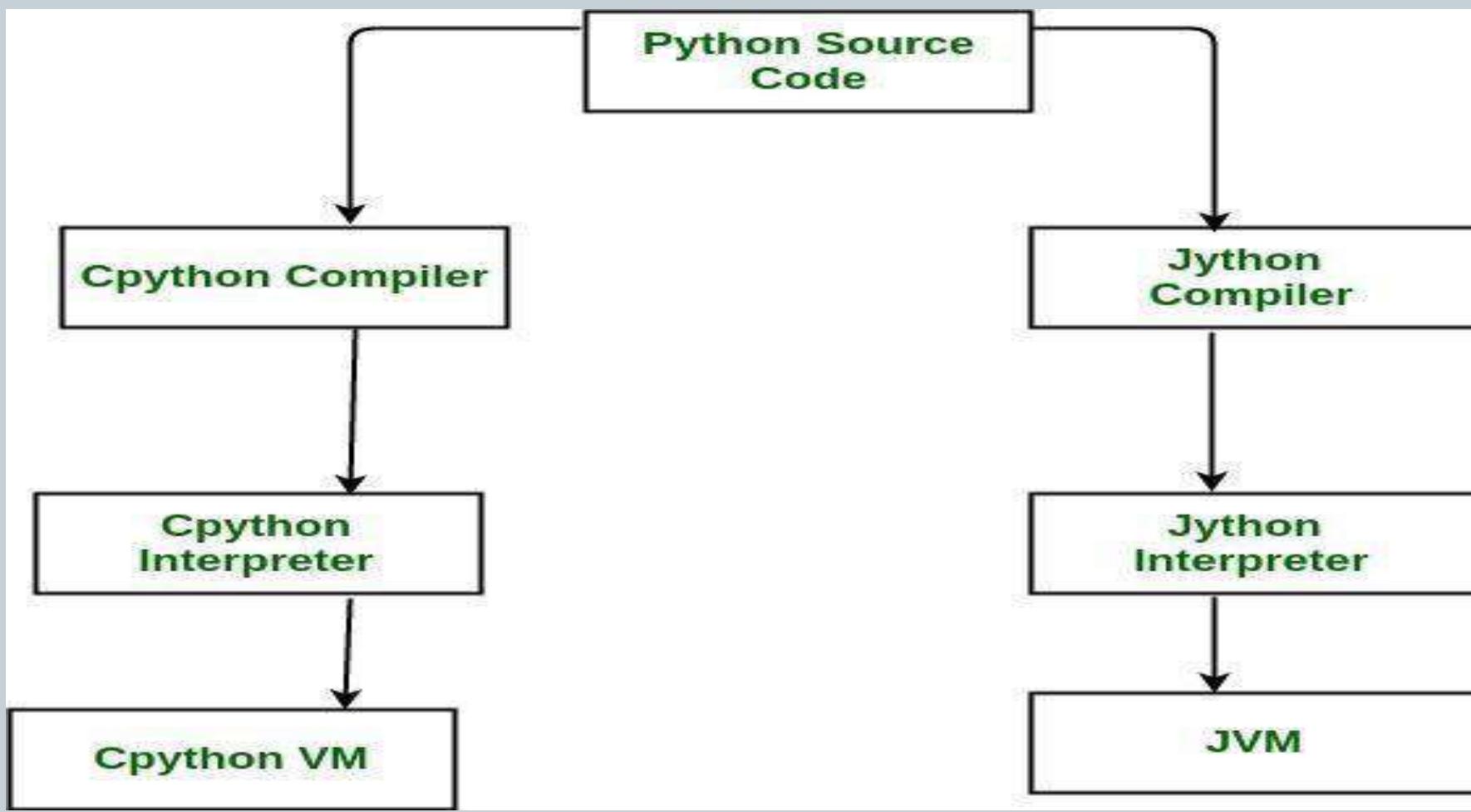
The **Jython** system (originally known as **JPython**) is an alternative implementation of the **Python** language, targeted for integration with the **Java** programming language.

Jython compiles **Python** source code to **Java bytecode** and then sends this **bytecode** to the **Java Virtual Machine** (JVM).

Because **Python** code is translated to **Java byte code**, it looks and feels like a true Java program at runtime.



Jython





IronPython



A third implementation of **Python**, and newer than both **CPython** and **Jython** is **IronPython**

IronPython is designed to allow **Python** programs to integrate with applications coded to work with **Microsoft's .NET Framework** for **Windows**.

Similar to **Jython**, it uses **.Net Virtual Machine** which is called as **Common Language Runtime**



PyPy



PyPy is an implementation of the **Python** programming language written in **Python**.

It uses a special compiler called **JITC** (just-in-time compilation).

PyPy adds **JITC** to **PVM** which makes the **PVM** more efficient and **fast** by converting **bytecode** into **machine code** in much **more efficient way** than the **normal interpreter**.

Downloading And Installing Python



Python's downloading and **installation** is fairly **easy** and is almost same as any other software.

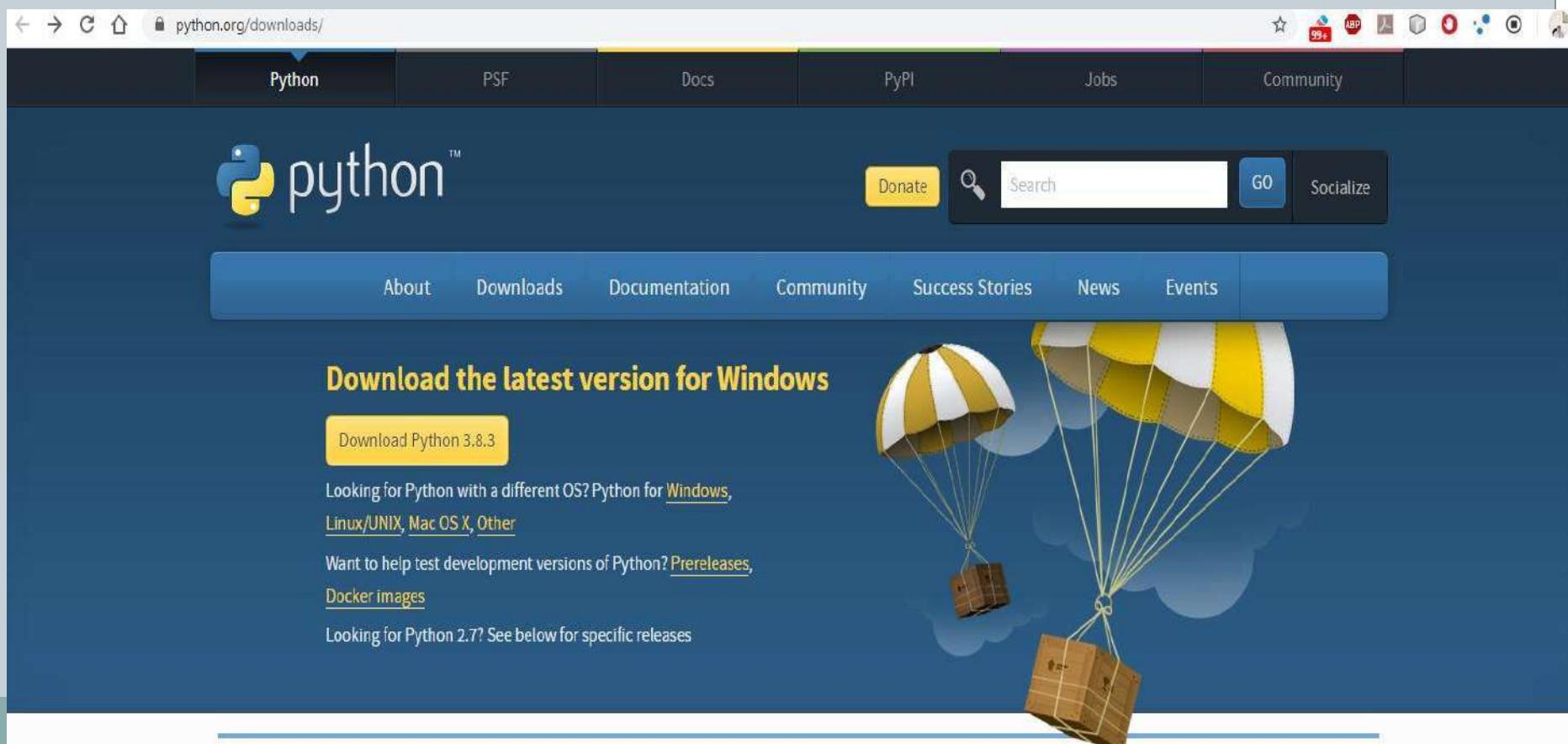
We can download everything we need to get started with **Python** from the **Python** website called <http://www.python.org/downloads>

The website should automatically detect that we're using **Windows** and present the links to the **Windows installer**.

Downloading And Installing Python



If you have **Windows 32 bit** then download the installer by clicking on the button **Download Python 3.8.3**



A screenshot of the Python.org Downloads page. The page has a dark blue header with the Python logo and navigation links for Python, PSF, Docs, PyPI, Jobs, and Community. Below the header is a main menu with links for About, Downloads, Documentation, Community, Success Stories, News, and Events. A prominent yellow button labeled "Download Python 3.8.3" is centered above a section titled "Download the latest version for Windows". This section also includes links for "Windows", "Linux/UNIX", "Mac OS X", and "Other". Below these are links for "Prereleases" and "Docker images". A note for Python 2.7 users is present. To the right of this text is a cartoon illustration of two boxes descending from the sky on parachutes. The background features a large white circle with a blue outline.

python.org/downloads/

Python PSF Docs PyPI Jobs Community

Donate Search GO Socialize

About Downloads Documentation Community Success Stories News Events

Download the latest version for Windows

Download Python 3.8.3

Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [Mac OS X](#), [Other](#)

Want to help test development versions of Python? [Prereleases](#), [Docker images](#)

Looking for Python 2.7? See below for specific releases



Downloading And Installing Python



But if you have windows 64 bit then scroll down and select **python 3.8.3** from the list

python.org/downloads/

Looking for a specific release?

Python releases by version number:

Release version	Release date	Click for more
Python 3.8.3	May 13, 2020	Download Release Notes
Python 3.8.3rc1	April 29, 2020	Download Release Notes
Python 2.7.18	April 20, 2020	Download Release Notes
Python 3.7.7	March 10, 2020	Download Release Notes
Python 3.8.2	Feb. 24, 2020	Download Release Notes
Python 3.8.1	Dec. 18, 2019	Download Release Notes
Python 3.7.6	Dec. 18, 2019	Download Release Notes
Python 3.6.10	Dec. 18, 2019	Download Release Notes

[View older releases](#)

Licenses

All Python releases are Open Source.
Historically, most, but not all,
Python releases have also been GPL-

Sources

For most Unix systems, you must
download and compile the source
code. The same source code archive

Alternative

Implementations
This site hosts the "traditional"
implementation of Python

History

Python was created in the early
1990s by Guido van Rossum at
Stichting Mathematisch Centrum in

Downloading And Installing Python



Now go to the **Files** section and select **windows x86-64 executable installer**

This will download the installer

A screenshot of a web browser window displaying the Python download page at python.org/downloads/release/python-383/. The title bar shows the URL. The page content includes a "Full Changelog" link and various download links for different operating systems and file types. A red box highlights the "Windows x86-64 executable installer" link, and a green box highlights the "Windows x86 executable installer" link.

Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		a7c10a2ac9d62de75a0ca5204e2e7d07	24067487	SIG
XZ compressed source tarball	Source release		3000cf50aaa413052aef82fd2122ca78	17912964	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	dd5e7f64e255d21f8d407f39a7a41ba9	30119791	SIG
Windows help file	Windows		4aeeebd7cc8dd90d61e7cidda9cb9422	8568303	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	c12ffe7f4c1b447241d5d2aedc9b5d01	8175801	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	fd2458fa0e9ead1dd9fb2370a42853b	27805800	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	17e989d2fecf7f9f13cf987825b695c4	1364136	SIG
Windows x86 embeddable zip file	Windows		8ee09403ec0cc2e89d43b4a4f6d1521e	7330315	SIG
Windows x86 executable installer	Windows		452373e2c467c14220efeb10f40c231f	26744744	SIG
Windows x86 web-based installer	Windows		fe72582bbca3dbe07451fd05ece1d752	1325800	SIG

Downloading And Installing Python



Open the downloads folder and run the file **python-3.8.3.exe** (if you are on 32 bit) or **python-3.8.3-amd64** (if you are on 64bit) by **right clicking** it and selecting **run as administrator**

Downloading And Installing Python



Once the installation is over you will get **SETUP WAS
SUCCESSFUL** message



Testing Python Installation



To verify that **Python** is installed and working correctly, do the following:

- Open the **command prompt**
- Type the command **python --version**

In the output we should see the **Python version number** as shown in the next slide



Testing Python Installation

To verify that **Python** is **installed** and **working correctly**, do the following:

A screenshot of a Windows Command Prompt window titled "cmd C:\Windows\system32\cmd.exe". The command "python --version" is entered, resulting in the output "Python 3.8.2". A yellow arrow points from the text "This is 3.8.2 but your Python version would be 3.8.3" to the displayed version number "3.8.2".

```
C:\Windows\system32\cmd.exe
C:\Users\Sachin>python --version
Python 3.8.2
This is 3.8.2 but your Python
version would be 3.8.3
C:\Users\Sachin>_
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 3



QUIZ- Test Your Skills

1. What is the correct syntax of **print** statement in Python 2.x ?

- A. print
- B. print()
- C. Print()
- D. Print

Correct Answer:A



QUIZ- Test Your Skills

2. What is the correct syntax of **print** function in Python 3.x ?

- A. print
- B. print()
- C. Print()
- D. Print

Correct Answer: B



QUIZ- Test Your Skills



3. Python is case sensitive

- A. False
- B. True

Correct Answer: B



QUIZ- Test Your Skills

4. Python is a compiled language or interpreted language ?

- A. Compiled Language
- B. Interpreted Language
- C. Both
- D. None Of The Above

Correct Answer:C



QUIZ- Test Your Skills

5. A Python code is normally smaller than the corresponding C language code

- A. True
- B. False

Correct Answer:A

QUIZ- Test Your Skills



6. A Python code runs faster than the corresponding C language code

- A. True
- B. False

Correct Answer: B



QUIZ- Test Your Skills

7. What kind of code Python compiler produces ?

- A. Machine Code
- B. Secret Code
- C. Source Code
- D. Byte Code

Correct Answer:D



QUIZ- Test Your Skills



8. What is CPython ?

- A. A Python Library
- B. Name Of Python Framework
- C. A Python language Implementation
- D. None Of The Above

Correct Answer:C

QUIZ- Test Your Skills



9. In CPython , the bytecode is converted to machine instruction set by
- A. PVM
 - B. VM
 - C. JVM
 - D. Bytecode Converter

Correct Answer:**A**

QUIZ- Test Your Skills



10. Python 3 is backward compatible with Python 2

- A. True
- B. False

Correct Answer: B

QUIZ- Test Your Skills



11. Support for Python 2 has ended on

- A. 31-Jan-2019
- B. 1-Jan-2020
- C. 31-Dec-2018
- D. 31-Dec-2019

Correct Answer: B



QUIZ- Test Your Skills



12. Arrange the following in descending order of speed of execution

- A. CPython , PyPy, C
- B. PyPy, C,CPython
- C. CPython ,C, PyPy
- D. C,PyPy,CPython

Correct Answer: D

QUIZ- Test Your Skills



13. Which implementation of Python contains JITC ?

- A. CPython
- B. PyPy
- C. Jython
- D. IronPython

Correct Answer: B

QUIZ- Test Your Skills



14. What is the output of $10/4$ in Python 3?

- A. 2.0
- B. 2.5
- C. 2
- D. None Of The Above

Correct Answer: B



QUIZ- Test Your Skills



15. What is the output of `print “Python Rocks”` in Python 3?

- A. Python Rocks
- B. Python
- C. Syntax Error
- D. None Of The Above

Correct Answer:C



QUIZ- Test Your Skills

16. Which of the following is not a Python IDE ?

- A. PyCharm
- B. Cutie Pie
- C. Spyder
- D. Visual Studio Code

Correct Answer: B

QUIZ- Test Your Skills



17. What is NumPy?

- A. A library of Python for working with large and multidimensional arrays
- B. A library of Python for Artificial Intelligence
- C. A Python IDE
- D. None of the above

Correct Answer:A

QUIZ- Test Your Skills



18. Python is a statically typed language

- A. True
- B. False

Correct Answer: B

These Notes Have Python _ world _ In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Today's Agenda



Developing First Python Code

- **What Is Python Shell ?**
- **Using Python Shell**
- **Writing Python Script Using Notepad**
- **Running Python Script**
- **How To View The Bytecode File ?**

Two Ways Of Interacting With Python



- In the last chapter, we have installed **Python**.
- Now let's start using it
- We can use **Python** in **two** modes:
 - **Interactive Mode**
 - **Script Mode**

Two Ways Of Interacting With Python



- In **Interactive Mode**, **Python** waits for us to enter command.
- When we type the command, **Python** interpreter goes ahead and executes the command, and then it waits again for our next command.
- In **Script mode**, **Python** interpreter runs a program from the source file.



The Interactive Mode

- **Python interpreter** in **interactive mode** is commonly known as **Python Shell**.
- To start the **Python Shell** enter the following command in the start menu search box:**python**
- Doing this will activate the **Python Shell** and now we can use it for running python statements or commands



The Interactive Mode



```
Python 3.8 (64-bit)
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



The Interactive Mode

- What we have seen on the previous slide is called **Python Shell**.
- `>>>` is known as **python prompt** or **prompt string**, and it simply means that **Python Shell** is ready to accept our commands.
- **Python Shell** allows us to type Python code and see the result immediately.



The Interactive Mode



- In technical jargon this is also known as **REPL** which stands for **Read-Eval-Print-Loop**.
- Whenever we hear **REPL** we should think of an environment which allows us to quickly test code snippets and see results immediately, just like a **Calculator**.
- Some examples of commands / code snippets to be run on shell are shown in the next slide

The Interactive Mode



```
C:\Windows\system32\cmd.exe - python
C:\Users\Sachin>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> 7+4
11
>>> 6/2
3.0
>>> 8*3
24
>>> 3/4
0.75
>>> _
```



The Interactive Mode

- Not only mathematical calculations , we also can run some basic python commands on **Python Shell**
- For example: Type the following command:
 - print("Hello Bhopal")**
 - And you will get the text displayed on the **Shell**

```
>>> print("Hello Bhopal")
Hello Bhopal
```



The Interactive Mode

- We must remember that **Python** is also a case sensitive language like **Cor C++**.
- So the function names must appear in lower case , otherwise **Python** generates **Error** , as shown below

```
>>> Print("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
>>>
```

Getting Help From Shell



- We can also **use help** on **Shell** for getting information on **Python** topics.
- To do this we need to type **help()** on the prompt string on the **Shell**

```
>>> help()
Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

Getting Help From Shell



- Now we get help on various **Python** topics .
- For example to get a list of all the available keywords in **Python**, we can write the command “**keywords**”

```
help> keywords
```

Here is a list of the Python keywords. Enter any keyword to get more help.

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

```
help>
```



Quitting Help

- To come out of help mode , we just have to strike **ENTER** key on the prompt

```
help>  Just Strike ENTER key here
```

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.

```
>>> ■
```



Quitting Shell



- To come out of **Python Shell**, we have to type the command **exit()** or **quit()** on the prompt string

```
>>> exit()
```

```
c:\Users\Sachin>
```



The Script Mode

Python Shell is great for testing small chunks of code but there is one problem - the statements we enter in the Python shell are not saved anywhere.

So if we want to execute same set of statements multiple times we will have to write them multiple times which is a difficult task.

In this case it is better to write the code in a **File** , **Save it** and then **Run it**

This is called **ScriptMode**

The Script Mode



In this mode we take following steps for developing and running a **Python** code:

- **Write the source code**

- **Compile it**
(Generation of bytecode)

- **Run it**
(Interpretation/Execution of the bytecode)

As discussed previously , **step 2** is **hidden from the programmer** and is **internally performed** by **Python** itself , so we just have to perform **step 1** and **step 3**



The Script Mode



For this do the following:

- Create a **directory** by any name at any location . I am creating it by the name of “**My Python Codes**” in **D:** drive .

- Open **notepad** and type the code as shown in the next slide in the file.



The Script Mode

```
print("Hello User")  
print("Python Rocks")
```

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

The Script Mode



- Now save this file by the name **firstcode.py** in the folder “**My Python Codes**” in **D:** drive .
- Remember the file can have any name but the extension must compulsorily be **.py**
- Now open the command prompt , move to the folder **My Python Codes** and type the following command:

python firstcode.py

- Here “**python**” is the name of Python’s interpreter which will run the program **firstcode.py**



The Script Mode



```
C:\Windows\system32\cmd.exe
D:\My Python Codes>python firstcode.py
Hello User
Python Rocks
D:\My Python Codes>_
```

D:\My Python Codes>python firstcode.py  **Command to run the code**

Hello User
Python Rocks  **Output of the code**

What Happened In Background?



- When a **Python** program **executes**, a **bytecode compiler** translates source code into **bytecode**.
- This **bytecode** is stored in **RAM** and **not visible** to us.
- After the **bytecode** is produced, it is then processed by the **PVM**(**Python Virtual Machine** a.k.a **Python Runtime** or **Python interpreter**).
- So the **Python compiler** produces **bytecode** in bulk while the **Python interpreter** inside **PVM** performs **line-by-line** execution of the **bytecode**.

What Happened In Background?

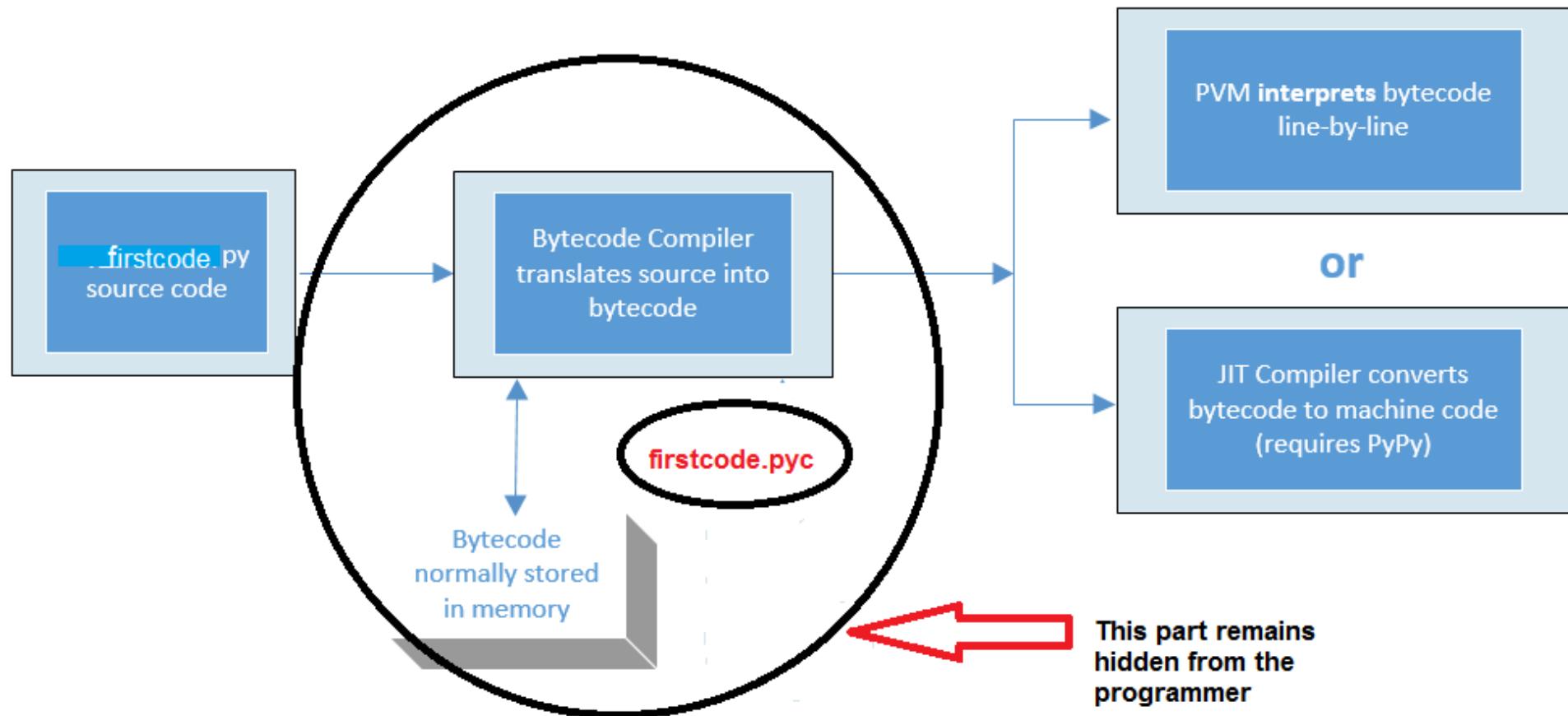


- So in our case the following sequence of events will take place:
 1. Programmer writes **the source code** by the name **firstcode.py**
 2. Then he/she **runs the program** using the command:
python firstcode.py
 3. The Python compiler internally generates **the bytecode file** called as **firstcode.pyc**
 4. Then the Python interpreter gets invoked and it reads this **bytecode file** and converts each **bytecode instruction** to the underlying operating system's instruction set and sends it for execution to the CPU
 5. Finally the programmer then sees the output

What Happened In Background?



Python Compilation & Interpretation



Can We See The Bytecode File ?



- Yes, we can force **Python** to save the **bytecode** file for us so that we view it.
- To do this we need to write the following command
python -m py_compile firstcode.py
- In the above command , we are using the switch **-m** , which is called **Module**.
- **Module** in **Python** are just like **header files of C/C++** language as it contains **functions ,classes and global variables**

Can We See The Bytecode File ?



- The module name in this command is **py_compile** and it generates **.pyc** file for the **.py** file.
- Now the Python compiler creates a separate folder called **__pycache__** for storing this **bytecode file**
- The name of the **bytecode file** is based on the **Python** implementation we are using
- Since we are using Cpython , so in our case the file name will be **firstcode.cpython-38.pyc**

Can We See The Bytecode File ?



- After we have created the **.pyc** file , the next step is to interpret it using **Python interpreter**
- The command for this will be:
python__pycache__\firstcode.cpython-36.pyc
- When we will run the above command the **Python interpreter** inside **PVM** will be invoked and will run the **bytecode** instructions inside the **firstcode.cpython-36.pyc** file



Can We See The Bytecode File ?



```
C:\Windows\system32\cmd.exe

D:\My Python Codes>python -m py_compile firstcode.py
D:\My Python Codes>python __pycache__\firstcode.cpython-36.pyc
Hello User
Python Rocks

D:\My Python Codes>
```

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



PYTHON

LECTURE 4



Today's Agenda



More About `print()` , IDLE, Error, Identifiers, Reserved Words

- **Introduction To Predefined Functions And Modules**
- **How `print()` function works ?**
- **How To Remove Newline From `print()` ?**
- **Introduction TOIDLE**
- **Types Of Errors In Python**
- **Rules For Identifiers**
- **Python Reserved Words**

Types Of Predefined Function Provided By Python



- Python has a very rich set of **predefined functions** and they are **broadly categorized** to be of **2 types**:
 - **Built In Functions**
 - **Functions Defined In Modules**



Built In Functions

- **Built in functions** are those functions which are always available for use .
- For example , `print()` is a **built-in function** which prints the given object to the standard output device (screen)
- As of version **3.8.5** , **Python** has **69 built-in function** and their list can be obtained on the following url :
<https://docs.python.org/3/library/functions.html>

What Is print() And How It Is Made Available To Our Program ?



Secure | <https://docs.python.org/3/library/functions.html>



Built-in Functions				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Functions Defined In Modules



A **Module** in **Python** is **collection of functions** and other **Python elements** which provide some extra functionality as compared to built in functions.

We can assume it just like a **header file** of **C/C++** language.

Python has 100s of built in **Modules** like **math , sys , platform** etc which prove to be very **useful** for a programmer

Functions Defined In Modules



For example , the module **math** contains a function called **factorial()** which can calculate and return the factorial of any number.

But to use a module we must first import it in our code using the syntax :

- **import <name of the module>**

For example: **import math**

Then we can call any function of this module by prefixing it with the module name

For example: **math.factorial(5)**

Functions Defined In Modules



```
>>> import math
>>> math.factorial(5)
120
>>>
```

```
>>> import platform
>>> platform.system()
'Windows'
```

How To Remove newline From print()



Let us revisit our **firstcode.py** file . The code was

```
print("Hello User")
print("Python Rocks")
```

The screenshot shows a Windows Command Prompt window titled 'cmd.exe' with the path 'C:\Windows\system32\cmd.exe'. The command 'D:\My Python Codes>python firstcode.py' is entered at the prompt. The output 'Hello User' and 'Python Rocks' is displayed below the command. Two red arrows point from the text 'Command to run the code' to the command line and from the text 'Output of the code' to the output text.

```
D:\My Python Codes>python firstcode.py
Hello User
Python Rocks
```

Command to run the code

Output of the code

How To Remove newline From print()



If we closely observe , we will see that the 2 messages are getting displayed on separate lines , even though we have not used any newline character.

This is because the function `print()` automatically appends a **newline character** after the message it is printing.

How To Remove newline From print()



If we do not want this then we can use the `print()` function as shown below:

```
print("Hello User",end="")
print("Python Rocks")
```

The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The command line shows the path 'D:\My Python Codes>' followed by the command 'python firstcode.py'. The output of the script is displayed below the command line, showing the two printed strings 'Hello User' and 'Python Rocks' on a single line, separated by a space. The command prompt then returns to the path 'D:\My Python Codes>'. The window has a standard Windows title bar and a black background.

```
D:\My Python Codes>python firstcode.py
Hello UserPython Rocks
D:\My Python Codes>
```

How To Remove newline From print()



The word **end** is called **keyword argument** in **Python** and it's default value is "**\n**".

But we have changed it to **empty string("")** to tell **Python** not to produce any newline.

Similarly we can set it to "**\t**" to generate tab or "**\b**" to erase the previous character



Some Examples

1.

```
print("Hello User",end="\t")
print("Python Rocks")
```

```
D:\My Python Codes>python firstcode.py
Hello User           Python Rocks
```

2.

```
print("Hello User",end="\b")
print("Python Rocks")
```

```
D:\My Python Codes>python firstcode.py
Hello UsePython Rocks
```



Introducing IDLE

When we install **CPython**, along with other tools we also get a lightweight **Integrated Development Environment** or **IDLE** for short.

The **IDLE** is a **GUI based IDE** for **editing** and **running Python programs**

IDLE has two main window types, the **Shell window** and the **Editor window**.

Shell window is same as **command shell** and **Editor window** is same as **notepad** but both have colorizing of **code**, **input**, **output**, and **error messages**.



Introducing IDLE

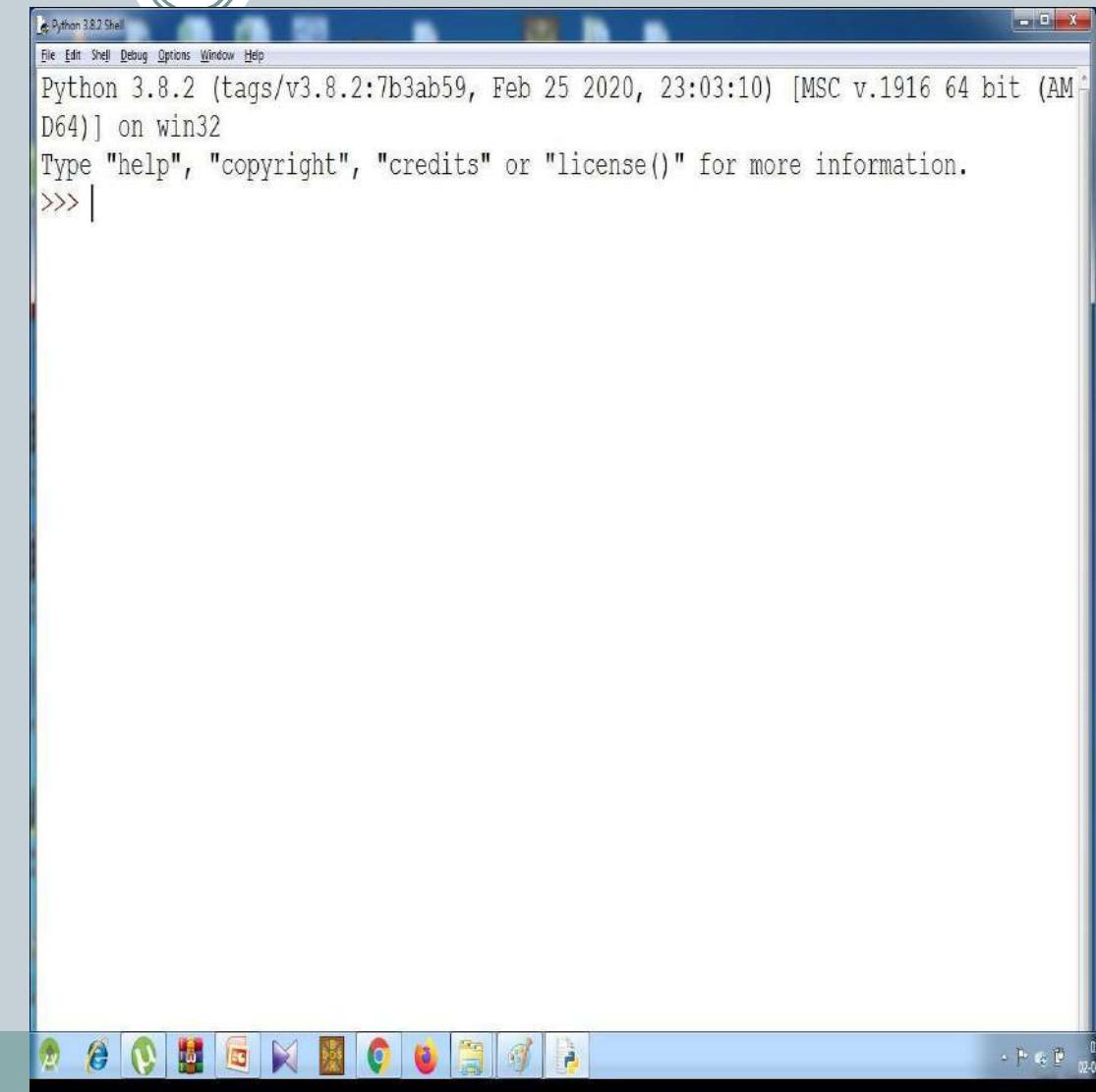
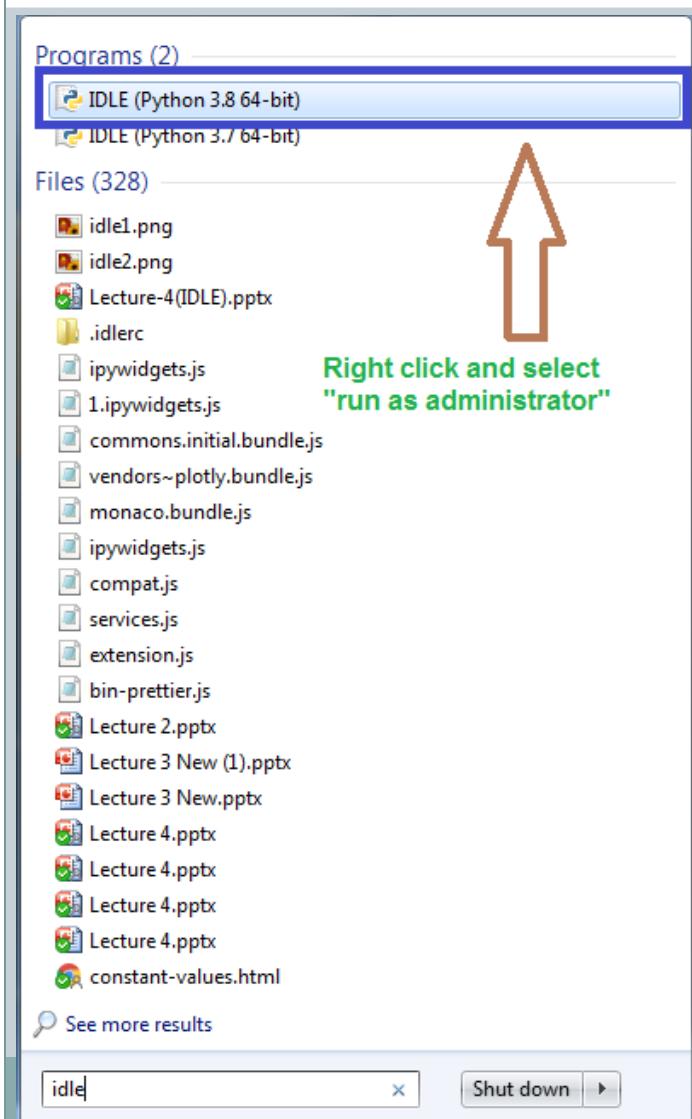


To start **IDLE** on Windows click the **Start Menu** and search "**IDLE**" or "**idle**".

Right Click **IDLE** as select **Run as administrator** and you will see a window as shown in the next slide



Opening IDLE





Using IDLE



This is again **Python Shell**, but a much more colourful as compared to the previous **Shell window**

Just type the commands, hit enter and it will display the result.



Using IDLE

Python 3.6.5 Shell

File Edit Shell Debug Options Window Help

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello User")
Hello User
>>> |
```

Ln: 5 Col: 4



Using IDLE's Editor Window

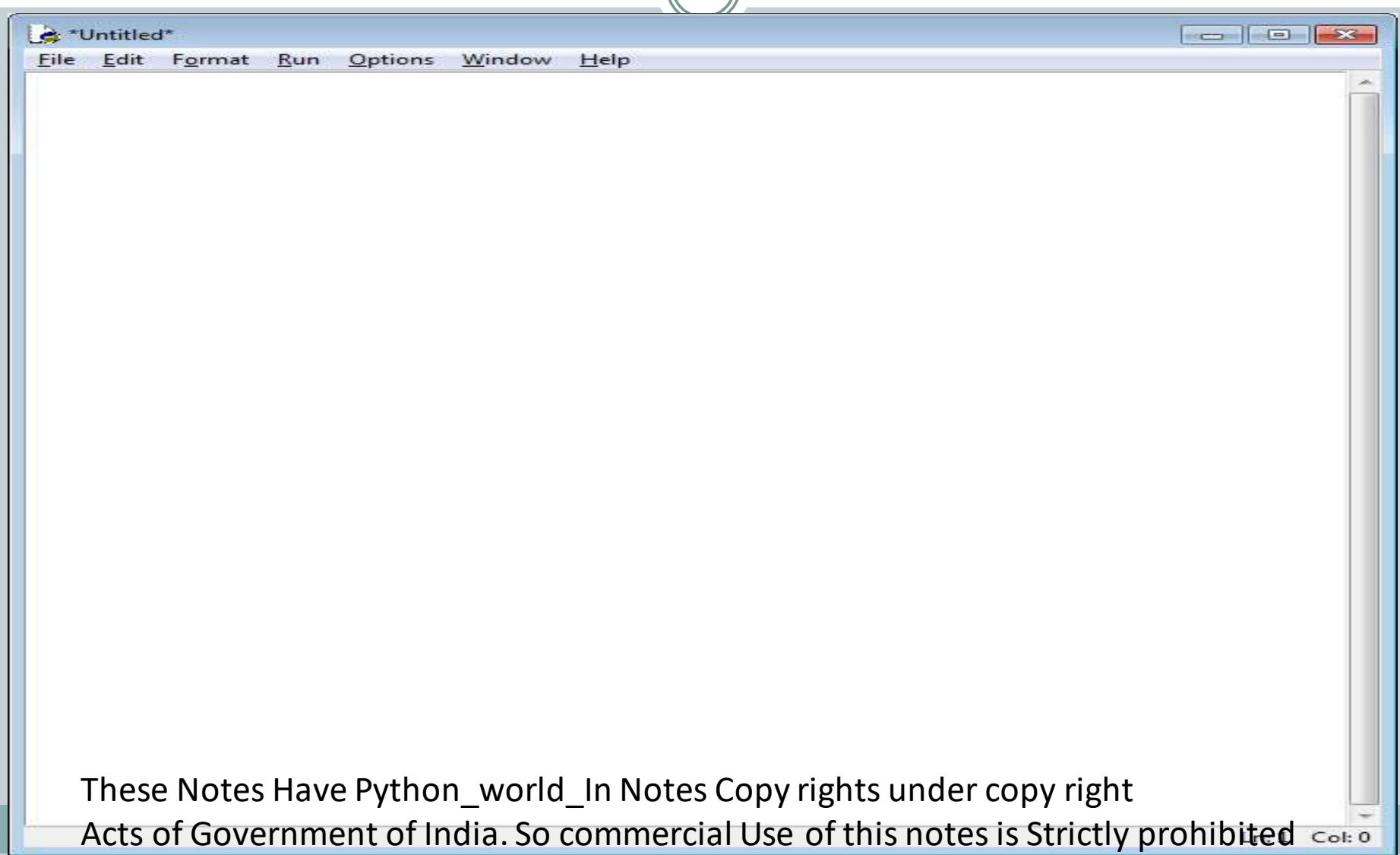
IDLE also has a **built-in text editor** to write Python programs.

To create a new program go to **File > New File**.

A new Untitled window will open. This window is a text editor where we can write programs.



Using IDLE's Editor Window



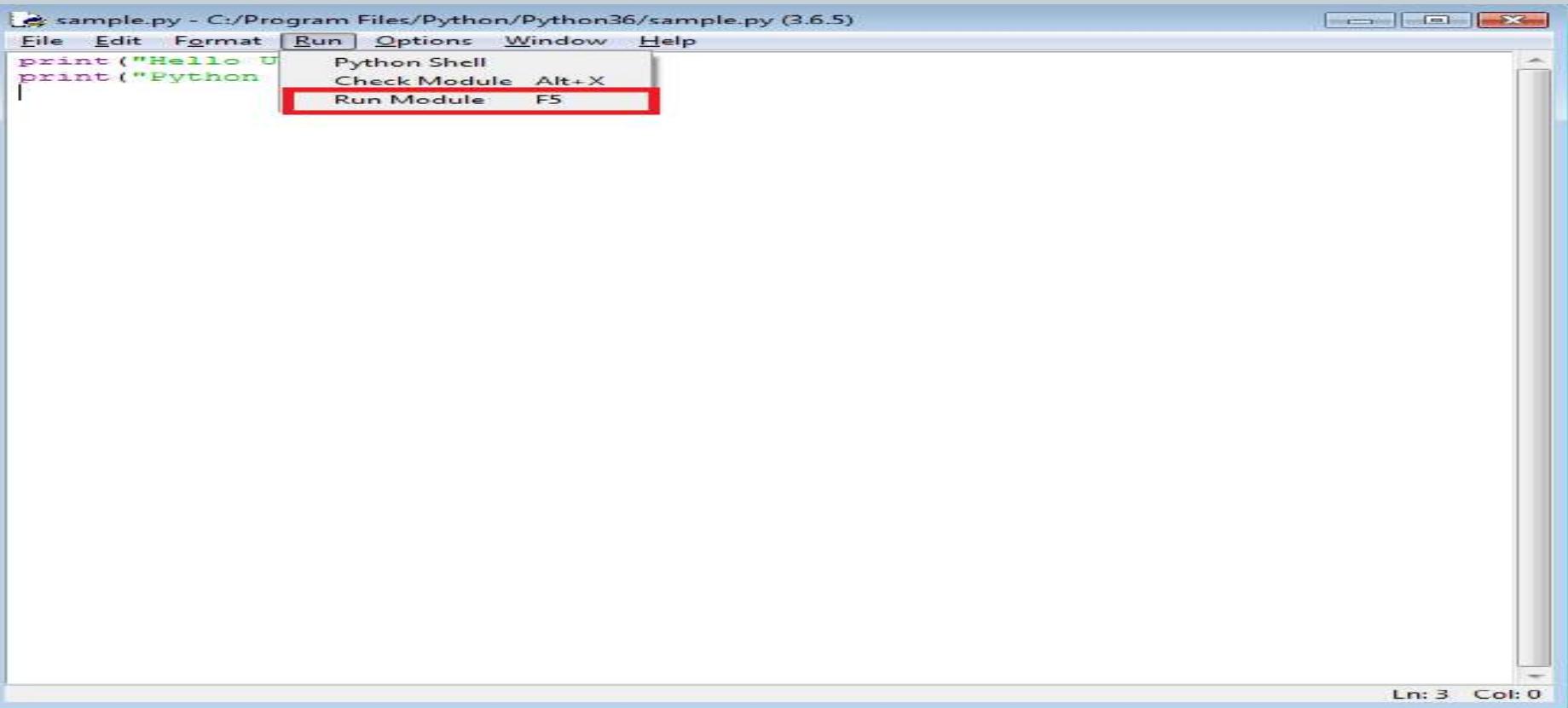
These Notes Have Python_world_In Notes Copy rights under copy right

Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Using IDLE's Editor Window

Save the file as **sample.py** and to run the program, Go to **Run > Run Module** or Hit**F5**.





Using IDLE's Editor Window

By doing this the **editor window** will move into the background, **Python Shell** will become active and we will see the output

The screenshot shows the Python 3.6.5 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the following text:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello User")
Hello User
>>>
=====
 RESTART: C:/Program Files/Python/Python36/sample.py =====
Hello User
Python Rocks!
>>> |
```

The status bar at the bottom right indicates Ln: 9 Col: 4.

Types Of Errors In Python



- Just like any other **programming language**, **Python** also has **2 kinds of errors**:
 - **Syntax Error**
 - **Runtime Error**



Syntax Error



- Syntaxes are **RULES OF A LANGUAGE** and when we break these rules , the error which occurs is called **Syntax Error**.
- Examples of **Syntax Errors** are:
 - **Misspelled keywords.**
 - **Incorrect use of an operator.**
 - **Omitting parentheses in a function call.**
 - **Unterminated strings**

And many other problems like this



Examples Of SyntaxError



```
>>> 1+
  File "<stdin>", line 1
    1+
      ^
SyntaxError: invalid syntax
>>>
```

```
>>> print("Hello)
  File "<stdin>", line 1
    print("Hello")
      ^
SyntaxError: EOL while scanning string literal
>>> -
```

RunTime Errors (Exceptions)

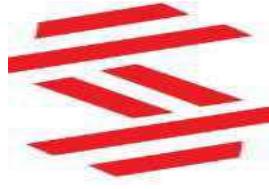


- As the name says, **Runtime Errors** are errors which occur while the **program** is **running**.
- As soon as **Python interpreter** encounters them it **halts the execution** of the program and **displays a message** about the probable cause of the problem.

RunTime Errors (Exceptions)



- They **usually occur** when **interpreter** counters an operation that is impossible to carry out and one such operation is **dividing a number by 0**.
- Since dividing a number by 0 is undefined , so ,when the interpreter encounters this operation it raises **ZeroDivisionError** as follows:



Example Of RunTimeError



```
>>> 10/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> ■
```



Rules For Identifiers

- **What is an identifier?**
 - **Identifier** is the **name** given to entities like **class**, **functions**, **variables**, **modules** and **any other object** in **Python**.
- **Rules for identifiers:**
 - **Identifiers** can be a combination of letters in **lowercase** (a to z) or **uppercase** (A to Z) or **digits** (0 to 9) or an **underscore** (_)
 - No **special character** except **underscore** is allowed in the name of a variable



Rules For Identifiers



- It must compulsorily begin with an **underscore** (_) or a **letter** and **not with a digit**. Although after the first letter we can have as many digits as we want. So **1a** is **invalid**, while **a1** or **_a** or **_1** is a **valid name** for an identifier.

```
>>> a_=10
>>> _a=10
>>> _1=10
>>> 1_=10
File "<stdin>", line 1
 1_=10
 ^
SyntaxError: invalid token
```



Rules For Identifiers



- Identifiers are case sensitive , so **pi** and **Pi** are two different identifiers.

```
>>> pi=3.14
>>> print(Pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Pi' is not defined
```



Rules For Identifiers



- Keywords cannot be used as identifiers

```
>>> if=15
      File "<stdin>", line 1
          if=15
          ^
SyntaxError: invalid syntax
```

- Identifier can be of any length.



Rules For Reserved Words

- **What is a Reserved Word?**
 - A word in a programming language which has a fixed meaning and cannot be redefined by the programmer or used as identifiers
- **How many reserved words are there in Python ?**
 - Python contains **35 reserved words** or **keywords**
 - The list is mentioned on the next slide
 - We can get this list by using **help()** in **Python Shell**

Rules For Reserved Words



These **35 keywords** are:

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

Some Important Observations:

1. Except **False , True and None** all the other **keywords** are in **lowercase**
2. We don't have **else if** in **Python** , rather it is **elif**
3. There are no **switch** and **do-while** statements in **Python**



python

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 5



Today's Agenda



Data Types

- Basic Data Types In Python
- Some Very Important Points To Remember
- Numeric Types
- Different Types Of Integers
- Converting Between Int Types



Basic Data Types In Python

- Although a **programmer is not allowed to mention the data type** while **creating variables** in his program in **Python**, but **Python internally allots different data types** to variables **depending on** their **declaration style** and **values**.
- Overall **Python** has **14 data types** and these are classified into **6 categories**.



Basic Data Types In Python

- These **categories** are:
 - **Numeric Types**
 - **Boolean Type**
 - **Sequence Types**
 - **Set Types**
 - **Mapping Type**
 - **None Type**
- **Given** on the **next slide** are the names of **actual data types** belonging to the above mentioned **categories**



Basic Data Types In Python

Numeric Type	Boolean Type	Sequence Type	Set Type	Mapping Type	None Type
<code>int</code>	<code>bool</code>	<code>str</code>	<code>set</code>	<code>dict</code>	<code>NoneType</code>
<code>float</code>		<code>list</code>	<code>frozenset</code>		
<code>complex</code>		<code>bytes</code>			
		<code>bytearray</code>			
		<code>tuple</code>			
		<code>range</code>			



Some Very Important Points

- Before we explore more about these data types , let us understand following important points regarding Python's data types:

1. DATA TYPES IN PYTHON ARE DYNAMIC
2. SIZE OF THE DATA TYPE IS ALSO DYNAMICALLY MANAGED
3. DATA TYPES ARE UNBOUNDED



Some Very Important Points

1. DATA TYPES IN PYTHON ARE DYNAMIC

- The term **dynamic** means that we can assign **different values** to the **same variable** at **different points** of time.
- Python will **dynamically change** the type of variable as per the value given.



Some Very Important Points

```
>>> a=10  
>>> print(a)  
10  
>>> type(a)  
<class 'int'>  
>>> a="sachin"  
>>> print(a)  
sachin  
>>> type(a)  
<class 'str'>  
>>> a=1.5  
>>> print(a)  
1.5  
>>> type(a)  
<class 'float'>  
>>>
```

type() is a built –in function and it returns the **data type** of the variable

Another important observation we can make is that in Python **all the data types are implemented as classes and all variables are objects**



Some Very Important Points

2. SIZE OF THE DATA TYPE IS ALSO DYNAMICALLY MANAGED

- In **Python** the size of **data types** is **dynamically managed**
- Like **C/C++/Java** language , variables in **Python** are **not of fixed size**.
- **Python** makes them **as big as required** on demand
- There is **no question** of **how much memory** a **variable uses** in **Python** because **this memory increases as per the value being assigned**



Some Very Important Points

- Python starts with **initial size** for a **variable** and then **increases its size** as needed up to the **RAM limit**
- This **initial size** for **int** is **24 bytes** and then **increases** as the **value is increased**
- If we **want to check** the **size** of a **variable**, then Python provides us a **function** called **getsizeof()** .
- This **function** is available in a **module** called **sys**



Some Very Important Points



```
>>> import sys  
>>> sys.getsizeof(0)  
24  
>>> sys.getsizeof(1)  
28  
>>> sys.getsizeof(123456789123456789123456789123456789)  
40  
>>>
```



Some Very Important Points

3. DATA TYPES ARE UNBOUNDED

- **Third important rule** to **remember** is that , in **Python** data types like **integers** don't have any range i.e. **they are unbounded**
- Like C /C++ /Java they don't have max or min value
- So an **int** variable can store **as many digits as we want.**



Numeric Types In Python

- As previously mentioned , Python supports 3 numeric types:
 - **int**: Used for storing **integer numbers** without any **fractional part**
 - **float**: Used for storing **fractional numbers**
 - **complex**: Used for storing **complex numbers**



Numeric Types In Python

- EXAMPLES OF `int` TYPE:

`a=10`

`b=256`

`c=-4`

`print(a)`

`print(b)`

`print(c)`

Output:

`10`

`256`

`-4`



Numeric Types In Python

- **DIFFERENT WAYS OF REPRESENTING `int` IN PYTHON:**
 1. As **decimal number**(base 10)
 2. As **binary number**(base 2)
 3. As **octal number**(base 8)
 4. As **hexadecimal number**(base 16)



Numeric Types In Python

- **REPRESENTING `int` AS DECIMAL (base 10):**
1. This is the **default way** of **representing integers**
 2. The term **base 10** means , 10 digits from **0 to 9** are allowed
 3. **Example:**

`a=25`



Numeric Types In Python

- REPRESENTING `int` AS BINARY(`base 2`) :
 1. We can **represent numeric values** as **binary values** also
 2. The term **base 2** means , only **2 digits** from **0 and 1** are allowed
 3. **But** we need to **prefix the number** with **0b** or **0B** , otherwise **Python** will take it to be a **decimal number**



Numeric Types In Python

```
>>> a=101  
>>> print(a)
```

101

```
>>> a=0b101  
>>> print(a)
```

5

Python is considering 101 as 101 only and not binary of 5

Now , Python will consider it as a binary value , since it has a prefix of 0b

Some Very Important Observation



1. For **representing binary value** it is **compulsory** to prefix the number with **0b** or **0B**.
2. **Although** we can assign **binary value** to the variable but when we display it we always get output in **decimal number system** form.

```
>>> a=0b101  
>>> print(a)  
5
```

Some Very Important Observation



3. We **cannot** provide any other digit except **0** and **1** while giving **binary value** , otherwise **Python** will generate **syntax error**.

```
>>> a=0b123
      File "<stdin>", line 1
        a=0b123
              ^
SyntaxError: invalid syntax
          0b123
```

Some Very Important Observation



4. We can provide **negative** values in **binary number system** also by prefixing **ob** with **-**.

```
>>> a=-0b101  
>>> print(a)  
-5
```



Numeric Types In Python

- REPRESENTING `int` AS OCTAL (base 8) :
 1. We can **represent numeric values** as **octal values** also
 2. The term **base 8** means , **only 8 digits** from **0 to 7** are allowed
 3. But we need to **prefix the number** with **zero** followed by **small o** or **capital O** i.e. either **oo** or **oO** , otherwise **Python** will take it to be a **decimal number**

```
>>> a=0o101  
>>> print(a)  
65
```



Numeric Types In Python

4. We **cannot provide** any other **digit** except **0 , 1 , 2 , 3 , 4 , 5 , 6** and **7** while giving **octal value** , otherwise **Python** will generate **syntax error**.

```
>>> a=0o181
      File "<stdin>", line 1
          a=0o181
                  ^
SyntaxError: invalid syntax
```



Numeric Types In Python

5. Just like **binary number system**, we can **provide negative values** in **octal number system** also by prefixing **0O** with -

```
>>> a=-00101  
>>> print(a)  
-65
```



Numeric Types In Python

- REPRESENTING `int` AS HEXADECIMAL (base 16) :
 1. We can **represent numeric values** as **hexadecimal values** also
 2. The term **base 16** means, **only 16 digits** from **0** to **9**, **a** to **f** and **A** to **F** are allowed
 3. But we need to **prefix the number** with **zero** followed by **small x** or **capital X** i.e. either **ox** or **oX**, otherwise Python will take it to be a **decimal number**

```
>>> a=0x101  
>>> print(a)  
257
```



Numeric Types In Python

4. We **cannot provide** any **other value** except the **digits** and **characters** from **A** to **F** while giving **hexadecimal value** , otherwise **Python** will generate **syntax error**.

```
>>> a=0xabcd
>>> print(a)
43981
>>> a=0xefgh
File "<stdin>", line 1
  a=0xefgh
          ^
SyntaxError: invalid syntax
```



Numeric Types In Python

5. Just like other **number systems**, we can provide negative values in **hexadecimal number system** also by prefixing **0x** with -

```
>>> a=-0xabcd  
>>> print(a)  
-43981
```



Base Conversion Functions

- We know that **Python** allows us to **represent integer values** in **4 different forms** like **int** , **binary** , **octal** and **hexadecimal**
- Moreover it also allows us to **convert one base type to another base type** with the help of certain **functions**.
- These **functions** are:
 - **bin()**
 - **oct()**
 - **hex()**



The `bin()` Function

- The `bin()` function **converts and returns** the **binary equivalent** of a **given integer**.
- **Syntax :** `bin(a)`
- **Parameters :** `a` : an integer to convert . This value can be of type **decimal** , **octal** or **hexadecimal**
- **Return Value :** A **string** representing **binary value**



The bin() Function

- Some Examples:

1. Converting decimal base to binary

```
>>> bin(25)  
'0b11001'
```

2. Converting octal base to binary

```
>>> bin(0o25)  
'0b10101'
```



The bin() Function

- Some Examples:

3. Converting hexadecimal base to binary

```
>>> bin(0x25)  
'0b100101'
```

4. Error if the value passed is not an integer

```
>>> bin("bhopal")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object cannot be interpreted as an integer
```



The oct() Function

- The **oct()** function **converts and returns** the octal equivalent of a **given integer**.
- **Syntax :** **oct(a)**
- **Parameters :** **a** : an integer to convert . This value can be of type **decimal** , **binary** or **hexadecimal**
- **Return Value :** A **string** representing **octal value**



The oct() Function

- Some Examples:

1. Converting decimal base to octal

```
>>> oct(25)  
'0o31'
```

2. Converting binary base to octal

```
>>> oct(0b101)  
'0o5'
```



The oct() Function

- Some Examples:

3. Converting hexadecimal base to octal

```
>>> oct(0x101)  
'0o401'
```

4. Error if the value passed is not an integer

```
>>> oct("hello")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object cannot be interpreted as an integer
```



The hex() Function

- The **hex()** function **converts and returns** the **hexadecimal equivalent** of a **given integer**.
- **Syntax :** `hex(a)`
- **Parameters :** `a` : an integer to convert . This value can be of type **decimal** , **octal** or **bin**
- **Return Value :** A **string** representing **hexadecimal value**



The hex() Function

- Some Examples:

1. Converting decimal base to hexadecimal

```
>>> hex(10)  
'0xa'
```

2. Converting binary base to hexadecimal

```
>>> hex(0b101)  
'0x5'
```



The hex() Function

- Some Examples:

3. Converting octal base to hexadecimal

```
>>> hex(0o25)  
'0x15'
```

4. Error if the value passed is not an integer

```
>>> hex("hello")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object cannot be interpreted as an integer
```



These Notes Have Python_world_In Notes Copyrights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 6



Today's Agenda



More On Data Types

- The **float** Type
- The **complex** Type
- The **bool** Type
- The **str** Type



The float Data Type

- Python also supports **floating-point real values**.
- **Float values** are specified with a **decimal point**
- So **2.5** , **3.14** , **6.9** etc are all examples of **float** data type
- Just like **double** data type of **other languages** like **Java/C** , float in **Python** has a precision of **16 digits**



Some Examples



```
>>> a=2.5  
>>> print(a)  
2.5  
>>> type(a)  
<class 'float'>
```

```
>>> 10/3  
3.333333333333335  
>>> .
```

Some Important Points About float



- For **float**, we can only assign values in **decimal number system** and not in **binary**, **octal** or **hexadecimal number system**.

```
>>> a=0o12.3
      File "<stdin>", line 1
          a=0o12.3
          ^
SyntaxError: invalid syntax
>>> a=0x12.3
      File "<stdin>", line 1
          a=0x12.3
          ^
SyntaxError: invalid syntax
```

Some Important Points About float



- **Float values** can also be represented as **exponential** values
- **Exponential notation** is a **scientific notation** which is represented using **e** or **E** followed by an **integer** and it means to the **power of 10**

```
>>> a=3.5e4  
>>> a  
35000.0
```



The complex Data Type

- **Complex numbers** are written in the form, $x + yj$, where **x** is the **real part** and **y** is the **imaginary part**.
- **For example:** $4+3j$, $12+1j$ etc
- The letter **j** is called **unit imaginary number**.
- It denotes the value of $\sqrt{-1}$, i.e j^2 denotes **-1**



An Example



```
>>> a=2+3j
>>> print(a)
(2+3j)
>>> type(a)
<class 'complex'>
```

Some Important Points About complex Data Type



- For representing the **unit imaginary number** we are only allowed to use the letter **j** (both **upper** and **lower case** are **allowed**).
- Any other letter if used will **generate error**

```
>>> a=2+3i
      File "<stdin>", line 1
          a=2+3i
                  ^
SyntaxError: invalid syntax
```

Some Important Points About complex Data Type



- The letter **j**, should only appear in **suffix**, not in **prefix**

```
>>> a=2+j3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j3' is not defined
```

Some Important Points About complex Data Type



- The **real** and **imaginary** parts are allowed to be **integers** as well as **floats**

```
>>> a=1.5+2.6j
>>> print(a)
(1.5+2.6j)
```

Some Important Points About complex Data Type



- The **real part** can be specified in any **int** form i.e. **decimal**, **binary**, **octal** or **hexadecimal** but the **imaginary part** should only be in **decimal form**

```
>>> a=0b101+2j      Allowed!
>>> print(a)
(5+2j)               Remember ! Displaying will
                      be always in decimal form
```

```
>>> a=5+0b010j
      File "<stdin>", line 1
          a=5+0b010j
                  ^
SyntaxError: invalid syntax
```

Some Important Points About complex Data Type



- We can display **real** and **imaginary** part separately by using the attributes of **complex** types called "**real**" and "**imag**".

```
>>> a=2+5j
>>> print(a.real)
2.0
>>> print(a.imag)
5.0
```

- Don't think **real** and **imag** are functions , rather they are **attributes/properties** of **complex** data type



The bool Data Type

- In **Python**, to represent **boolean** values we have **bool** data type.
- The **bool** data type can be one of two values, either **True** or **False**.
- We use **Booleans** in programming to make **comparisons** and to **control the flow** of the program.



Some Examples



```
>>> a=False  
>>> print(a)  
False
```

```
>>> a=False  
>>> type(a)  
<class 'bool'>
```

Some Important Points About bool



- **True** and **False** are **keywords**, so **case sensitivity** must be **remembered** while assigning them otherwise **Python** will give error

```
>>> a=false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'false' is not defined
```

Some Important Points About bool



- All **test conditions** in **Python** return the result as **bool** which could be either **True** or **False**

```
>>> a=10
>>> b=5
>>> print(a>b)
True
```

```
>>> x=15
>>> y=15
>>> print(x<y)
False
```

Some Important Points About bool



- To understand the next point , try to guess the output of the following:

a=True

b=False

c=a+b

print(c)

Output:

1

a=True

b=True

c=a+b

print(c)

Output:

2

a=False

b=False

c=a+b

print(c)

Output:

0

The above outputs make it clear that internally **Python** stores **True** and **False** as **integers** with the value **1** and **0** respectively



The str Data Type

- Just like any **other language**, in **Python** also a **String** is **sequence of characters**.
- **Python** does not have a **char data type**, unlike **C/C++** or **Java**
- We can use **single quotes** or **double quotes** to represent **strings**.
- However **Python** recommends to use **single quotes**



Some Examples



```
>>> name='Sachin'  
>>> print(name)  
Sachin
```

```
>>> name="Sachin"  
>>> print(name)  
Sachin
```

The data type used by **Python** internally for storing **Strings** is
str

```
>>> name="Sachin"  
>>> type(name)  
<class 'str'>
```

Some Important Points About Strings



- Unlike **C** language , **Python** does not uses **ASCII number system** for characters . It uses **UNICODE number system**
- **UNICODE** is a **number system** which supports much wider range of characters compared to **ASCII**
- As far as **Python** is concerned , it uses **UNICODE** to support **65536** characters with their numeric values ranging from **0** to **65535** which covers almost every spoken language in the world like **English , Greek , Spanish , Chinese , Japanese** etc

Some Important Points About Strings



To quote the unicode website they are atleast 61 different languages supported.

<http://www.lexilogos.com/keyboard/index.htm>

Какъо е Unicode? in Bulgarian (30 letters)
Što je Unicode? in Croatian (30 letters)
Co je Unicode? in Czech (48 letters)
Hvad er Unicode? in Danish(29 letters)
Wat is Unicode? in Dutch(26 letters)
𠁻𠁻𠁻 𠁻 𠁻 𠁻 𠁻 𠁻? in English (Deseret)
𠁻𠁻𠁻 𠁻 𠁻 𠁻 𠁻? in English (Shavian)
Kio estas Unikodo? in Esperanto(31 letters)
Mikä on Unicode? in Finnish(29 letters)
Qu'est ce qu'Unicode? in French
რა არის უნიკოდი? in Georgian
Was ist Unicode? in German
Τι είναι το Unicode; in Greek (Monotonic)
Τι είναι τὸ Unicode; in Greek (Polytonic)
תְּקִוָּן נִנְהַנְּ (Unicode)? in Hebrew
यूनिकोड क्या है? in Hindi
Mi az Unicode? in Hungarian
Hvað er Unicode? in Icelandic
Gini bù Yunikod? in Igbo
Que es Unicode? in Interlingua
Cos'è Unicode? in Italian
ユニコードとは? in Japanese
ಉನಿಕೋಡ್ ಎಂದರೆನು? in Kannada
유니코드에 대해서? in Korean
Kas tai yra Unikodas? in Lithuanian
Што е Unicode? in Macedonian
X'inhu l-Unicode? in Maltese
Unicode гэх мүйнэ? in Mongolian
युनिकोड के हो? in Nepali
Unicode, qu'es aquò? in Occitan
چیست بونی گند in Persian
Czym jest Unikod? in Polish
O que é Unicode? in Portuguese

Some Important Points About Strings



- Whenever we display a **string value** directly on **Python's shell** i.e. without using the function **print()**, **Python's shell** automatically encloses it in **single quotes**

```
>>> a="hello"  
>>> a  
'hello'
```

- However this does not happen when we use **print()** function to print a **string value**

```
>>> a="Hello"  
>>> print(a)  
Hello
```

Some Important Points About Strings



- If a **string starts** with **double quotes** , it must **end** with **double quotes** only .
- Similarly if it **starts** with **single quotes** , it must **end** with **single quotes** only.
- Otherwise **Python** will generate **error**

Some Important Points About Strings



```
>>> s="Welcome"  
>>> print(s)  
Welcome  
>>> s="Welcome'  
      File "<stdin>", line 1  
        s="Welcome'  
              ^  
SyntaxError: EOL while scanning string literal
```

Some Important Points About Strings



- If the string contains **single quotes** in between then it must be enclosed in **double quotes** and **vice versa**.
- **For example:**
- To print **Sachin's Python Classes**, we would write:
 - **msg= " Sachin's Python Classes "**
- Similarly to print **Capital of "MP" is "Bhopal"**,we would write:
 - **msg='Capital of"MP" is "Bhopal" '**

Some Important Points About Strings



```
>>> msg="Sachin's Python Classes"  
>>> print(msg)  
Sachin's Python Classes
```

```
>>> msg='Capital of "MP" is "Bhopal"'  
>>> print(msg)  
Capital of "MP" is "Bhopal"
```

Some Important Points About Strings



- How will you print **Let's learn "Python"** ?

A. "Let's learn "Python" "

B. 'Let's learn "Python" '

NONE!

Both will give error.

Correct way is to use either **triple single quotes** or **triple double quotes** or **escape sequence character **

msg=''' Let's learn "Python" '''

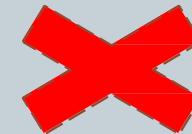
OR

msg='Let\'s learn "Python" '

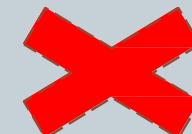
Some Important Points About Strings



```
>>> msg='Let's learn "Python" '
      File "<stdin>", line 1
          msg='Let's learn "Python"
                  ^
SyntaxError: invalid syntax
```



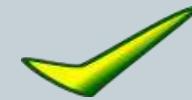
```
>>> msg="Let's learn "Python"""
      File "<stdin>", line 1
          msg="Let's learn "Python"
                  ^
SyntaxError: invalid syntax
```



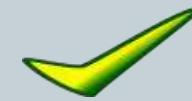
Some Important Points About Strings



```
>>> msg="""Let's learn "Python"""
>>> print(msg)
Let's learn "Python"
```



```
>>> msg='Let\'s learn "Python" '
>>> print(msg)
Let's learn "Python"
```



Some Important Points About Strings



- Another important use of **triple single quotes** or **triple double quotes** is that if our **string** extends up to more than 1 line then we need to enclose it in **triple single quotes** or **triple double quotes**

```
>>> msg="Sharma
  File "<stdin>", line 1
      msg="Sharma
          ^
SyntaxError: EOL while scanning string literal
```

```
>>> msg="""Sharma
... Computer
... Academy"""
>>> print(msg)
Sharma
Computer
Academy
```

Some Important Points About Strings



- We also can do the same thing by using `\n`, so using **triple quotes** or **triple double quotes** is just for improving readability

```
>>> msg="Sharma\nComputer\nAcademy"  
>>> print(msg)  
Sharma  
Computer  
Academy
```

Accessing Individual Characters In String



- In **Python**, all **Strings** are stored as **individual characters** in a **contiguous memory location**.
- Each **character** in this **memory location** is assigned an **index** which begins from **0** and goes up to **length -1**

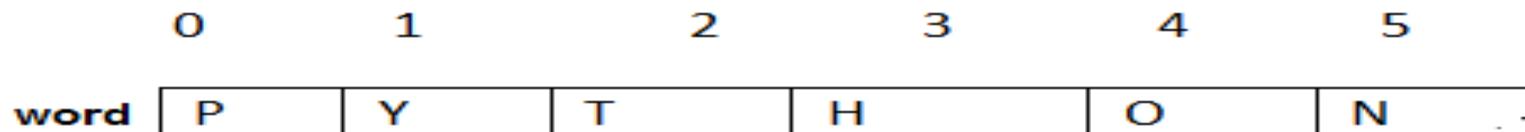
Accessing Individual Characters In String



- For example, suppose we write

word="Python"

- Then the internal representation of this will be



Accessing Individual Characters In String



- Now to access individual character we can provide this **index number** to the **subscript operator []**.

```
>>> word="Python"
>>> print(word[0])
P
>>> print(word[1])
y
>>> print(word[2])
t
```

Accessing Individual Characters In String



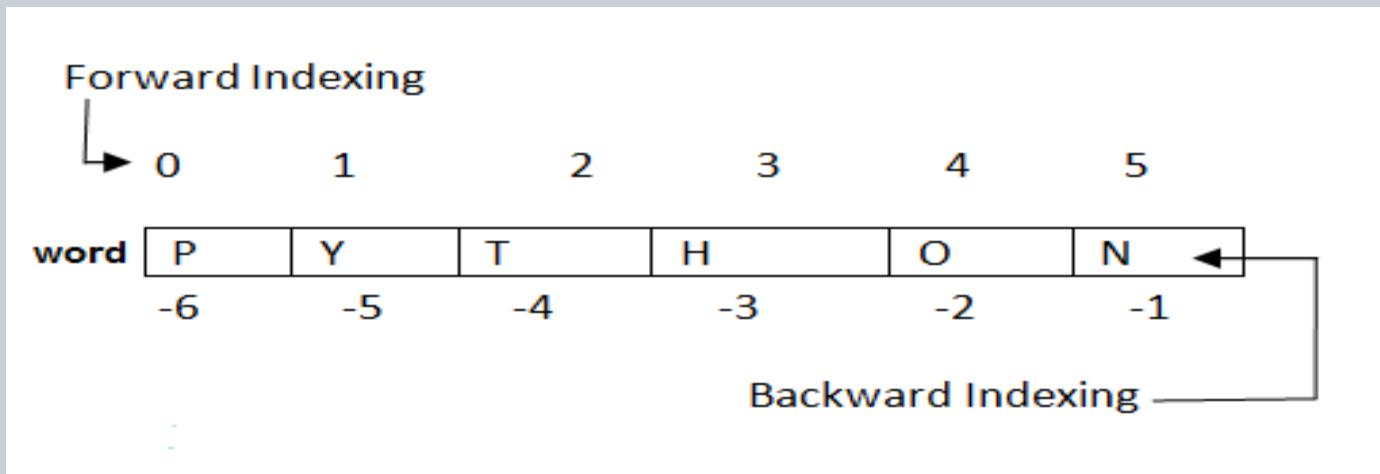
- However if we try to provide an **index number** beyond the given limit then **IndexError** exception will arise

```
>>> word="Python"
>>> print(word[7])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Accessing Individual Characters In String



- Not only this , **Python** even allows **negative indexing** which begins from the **end** of the **string**.
- So **-1** is the **index** of **last character** , **-2** is the **index** of **second last character** and so on.



Accessing Individual Characters In String



```
>>> word="Python"  
>>> print(word[-1])  
n  
>>> print(word[-2])  
o
```



These Notes Have Python _world _In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 7



Today's Agenda

- Concatenating Strings
- The Slice Operator In Strings
- Three Important String Functions
- Type Conversion



String Concatenation

- **Concatenation** means **joining** two or more **strings** together
- To **concatenate** strings, we use the **+** operator.
- **Keep in mind** that *when we work with numbers, + will be an operator for addition, but when used with strings it is a joining operator.*



String Concatenation

- Example:

`s1="Good"`

`s2="Morning"`

`s3=s1+s2`

`print(s3)`

- Example:

`s1="Good"`

`s2="Morning"`

`s3=s1+" "+s2`

`print(s3)`

- Output:

`Good Morning`

- Output:

`GoodMorning`



The Slicing Operator

- **Slicing** means **pulling out** a **sequence of characters** from a **string**.
- **For example** , if we have a string “**Industry**” and we want to extract the word “**dust**” from it , then in **Python** this is done using slicing.
- To slice a string , we use the operator[] as follows:
- **Syntax:** **s[x:y]**
 - **x** denotes the **start index** of slicing and **y** denotes the **end index** . But **Python** ends slicing at **y-1** index.



The Slicing Operator

- Example:

```
s="Industry"  
print(s[2:6])
```

- Example:

```
s="Welcome"  
print(s[3:6])
```

- Output:

dust

- Output:

com



The Slicing Operator

- **Example:**

```
s="Mumbai"  
print(s[0:3])
```

- **Example:**

```
s="Mumbai"  
print(s[0:10])
```

- **Output:**

Mum

- **Output:**

Mumbai



The Slicing Operator

- Example:

```
s="Python"  
print(s[2:2])
```

- Output:

- Example:

```
s="Python"  
print(s[6:10])
```

- Output:



The Slicing Operator

- Example:

```
s="welcome"  
print(s[1:])
```

- Output:

elcome

- Example:

```
s="welcome"  
print(s[:3])
```

- Output:

wel



The Slicing Operator

- **Example:**

```
s="welcome"  
print(s[:])
```

- **Output:**

welcome

- **Example:**

```
s="welcome"  
print(s[])
```

- **Output:**

Syntax Error



The Slicing Operator

- **Example:**

```
s="welcome"  
print(s[-4:-1])
```

- **Example:**

```
s="welcome"  
print(s[-1:-4])
```

- **Output:**

com

- **Output:**



Using Step Value

- **String slicing** can accept a **third parameter** also after the two index numbers.
- The **third parameter** is called **step value**.
- So the **complete syntax** of **slicing operator** is:

`s[begin:end:step]`

- **Step value** indicates *how many characters to move forward after the first character is retrieved* from the **string** and it's default value is **1** , but can be changed **as per our choice**.



The Slicing Operator

- For Example:

```
s="Industry"  
print(s[2:6])
```

```
s="Industry"  
print(s[2:6:2])
```

- Output:
`dust`

- Can also be written
as :

```
s="Industry"  
print(s[2:6:1])
```

- Output:
`dust`

- Output:
`ds`

Three Very Useful Functions/Methods Of String Data Type



- Python provides us some very useful functions/methods for performing various operations on String values.
- Following are these functions/methods:
 - len()
 - lower()
 - upper()

Three Very Useful Functions/Methods Of String Data Type



- **len()** : Returns **length** of the **String** passed as argument
- Syntax: **len(s)**

```
>>> city="Bhopal"
>>> print(len(city))
6
```

- **lower()** : Returns a **copy** of calling **String** object with all letters converted to **lowercase**
- Syntax: **s.lower()**

```
>>> s="Bhopal"
>>> print(s.lower())
bhopal
>>> print(s)
Bhopal
```

Three Very Useful Functions/Methods Of String Data Type



- **upper()** : Returns a copy of calling String object with all letters converted to uppercase
- **Syntax:** `s.upper()`

```
>>> s="Bhopal"
>>> print(s.upper())
BHOPAL
>>> print(s)
Bhopal
```



Comparing Strings



- We can use (`>`, `<`, `<=`, `>=`, `==`, `!=`) to **compare** two **strings**.
- **Python** compares string lexicographically i.e using **UNICODE** value of the characters.



Comparing Strings



- Suppose we have `str1` as " Indore " and `str2` as " India" and we write `print(str1>str2)` , then **Python** will print **True**. Following is the explanation
 - Now the first **two characters** from `str1` and `str2` (I and I) are **compared**.
 - As they are **equal**, the **second two characters** are **compared**.
 - Because they are also **equal**, the **third two characters** (d and d) are **compared**.
 - Since they also are **equal** , the **fourth pair** (o and i) is **compared** and there we get a **mismatch** .
 - Now because **o** has a greater **UNICODE** value than **i** so **Indore** is greater than **India** and so the answer is **True**



Comparing Strings



```
>>> str1="Indore"
>>> str2="India"
>>> print(str1==str2)
False
>>> print(str1>str2)
True
>>> print(str1==str1)
True
>>> print(str2==str2)
True
```



Type Conversion



- The process of **converting** the value of **one data type** (integer, string, float, etc.) to **another data type** is called **Type Conversion**.
- Python has **two** types of **type conversion**.
 - **Implicit Type Conversion**
 - **Explicit Type Conversion**



Implicit Conversion



- In **Implicit Type Conversion**, **Python** automatically converts one data type to another data type.
- This process doesn't need any programmer involvement.
- Let's see an example where **Python** promotes conversion of **int** to **float**.

Example Of Implicit Conversion



```
>>> a=10
>>> b=6.5
>>> c=a+b
>>> print(a)
10
>>> print(b)
6.5
>>> print(c)
16.5
>>> print(type(c))
<class 'float'>
```

- If we observe the above operations , we will find that **Python** has automatically assigned the data type of **c** to be **float** .
- This is because **Python** always converts **smaller data type** to **larger data type** to avoid the **loss of data**.



Another Example

```
>>> a=10
>>> b=True
>>> c=a+b
>>> print(a)
10
>>> print(b)
True
>>> print(c)
11
>>> print(type(c))
<class 'int'>
```

- Here also **Python** is automatically upgrading **bool** to type **int** so as to make the result sensible



Explicit Type Conversion

- There are some cases , where **Python** will not perform type conversion automatically and we will have to explicitly convert one type to another.
- Such **Type Conversions** are called **Explicit Type Conversion**
- Let's see an example of this



Explicit Type Conversion

Guess the output ?

a=10

b="6"

print(type(a))

print(type(b))

c=a+b

print(c)

print(type(c))

Output:

<class 'int'>

<class 'str'>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Why did the code fail?

The code **failed** because **Python** does not automatically convert **String** to **int**.

To handle such cases we need to perform **Explicit Type Conversion**

Explicit Type Conversion Functions In Python



- Python provides us **5 predefined functions** for performing **Explicit Type Conversion** for fundamental data types.
- These functions are :
 1. **int()**
 2. **float()**
 3. **complex()**
 4. **bool()**
 5. **str()**

These Notes Have Python _world _In Notes Copy rights under copy right
Acts of Government of India .So commercial Use of this notes is Strictly
prohibited



The `int()` Function

- **Syntax:** `int(value)`
- This function converts **value of any data type to integer** ,
with some special cases
- It returns an **integer** object converted from the given **value**



int() Examples

`int(2.3)`

Output:

`2`

`int(False)`

Output:

`0`

`int(True)`

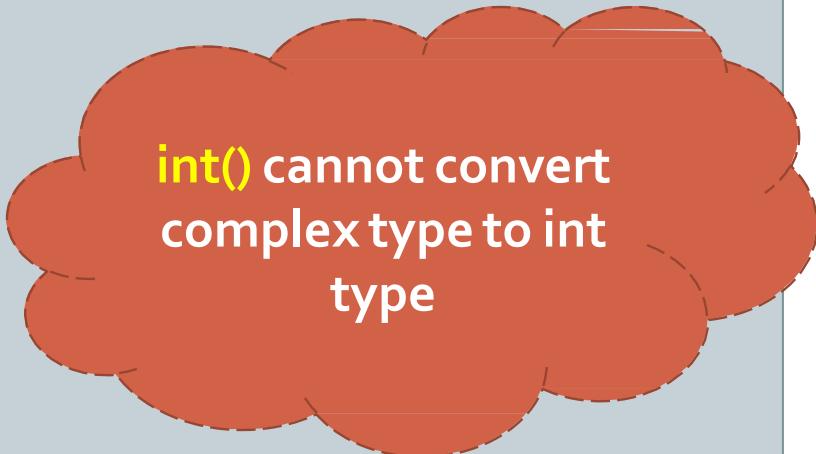
Output:

`1`

`int(3+4j)`

Output:

`TypeError: Can't convert complex to int`



`int()` cannot convert
complex type to int
type



int() Examples

```
int("25")
```

Output:

25

```
int("2.5")
```

Output:

ValueError: Invalid literal for int()

```
int("1010")
```

Output:

1010

```
int("0b1010")
```

Output:

ValueError: Invalid literal for int()

int() cannot accept anything other than digits in a string

int() cannot accept binary values as string

Solution To The Previous Problem



Can you solve this error now ?

a=10

b="6"

c=a+b

print(c)

Output:

TypeError

Solution:

a=10

b="6"

c=a+int(b)

print(c)

Output:

16



The **float()** Function

- **Syntax:** `float(value)`
- This function converts **value of any data type to float**,
with some special cases
- It returns a **float** object converted from the given **value**



float() Examples

`float(25)`

Output:

`25.0`

`float(False)`

Output:

`0.0`

`float(True)`

Output:

`1.0`

`float(3+4j)`

Output:

`TypeError: Can't convert complex to float`

`float()` cannot
convert complex
type to float type



float() Examples

`float("25")`

Output:

`25.0`

`float("2.5")`

Output:

`2.5`

`float("1010")`

Output:

`1010.0`

`float ("ob1010")`

Output:

`ValueError:Could not convert string to float`

`float("twenty")`

Output:

`ValueError:Could not convert string
to float`

float() cannot
accept any int value
other than base 10
as string



The **complex()** Function

- **Syntax:** `complex(value)`
- This function converts **value** of any data type to **complex**,
with some special cases
- It returns an **complex** object converted from the given
value



complex() Examples

`complex(25)`

Output:

`(25+0j)`

`complex(2.5)`

Output:

`(2.5+0j)`

`complex(True)`

Output:

`(1+0j)`

`complex(False)`

Output:

`0j`



complex() Examples

`complex("25")`

Output:

`(25+0j)`

`complex("2.5")`

Output:

`(2.5+0j)`

`complex("1010")`

Output:

`(1010+0j)`

`complex ("ob1010")`

Output:

`ValueError: complex() arg is a malformed string`

`complex("twenty")`

Output:

`ValueError: complex() arg is a
malformed string`

complex() cannot
accept any int value
other than base 10 as
string



The **bool()** Function

- **Syntax:** `bool(value)`
- This function converts **value of any data type to bool**,
using the standard truth testing procedure.
- It returns an **bool** object converted from the given **value**



The `bool()` Function

- What values are considered to be **false** and what values are **true** ?
- The following values are considered **false** in **Python**:
 - **None**
 - **False**
 - **Zero of any numeric type**. For example, `0`, `0.0`, `0+0j`
 - **Empty sequence**. For example: `()`, `[]`, `"."`.
 - **Empty mapping**. For example: `{}`
- All other values are **true**



bool() Examples

`bool(1)`

Output:

`True`

`bool(5)`

Output:

`True`

`bool(0)`

Output:

`False`

`bool(0.0)`

Output:

`False`



bool() Examples

`bool(0.1)`

Output:

`True`

`bool(0b101)`

Output:

`True`

`bool(0oooo)`

Output:

`False`

`bool(2+3j)`

Output:

`True`

`bool()` returns True if
any of the real or
imaginary part is non
zero . If both are zero
it returns False



bool() Examples

`bool(o+1j)`

Output:

`True`

`bool(o+oj)`

Output:

`False`

`bool("")`

Output:

`False`

`bool('A')`

Output:

`True`

`bool("twenty")`

Output:

`True`

`bool(' ')`

Output:

`True`

`bool()` returns `False`
for empty Strings
otherwise it returns
`True`



The **str()** Function

- **Syntax:** `str(value)`
- This function converts **any data type to string**, *without any special cases*
- It returns a **String** object converted from the given **value**



str() Examples

`str(15)`

Output:

`'15'`

`str(2.5)`

Output:

`'2.5'`

`str(2+3j)`

Output:

`'(2+3j)'`

`str(True)`

Output:

`'True'`



str() Examples

str(1)

Output:

'1'

str(5)

Output:

'5'

str(2.5)

Output:

'2.5'

str(True)

Output:

'True'

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 8



Today's Agenda

- **Variables And Memory Management**
 - How Variables In Python Are Different Than Other Languages ?
 - Immutable And Mutable
 - Python's Memory Management
 - The id() Function
 - The is Operator

Understanding Python Variables

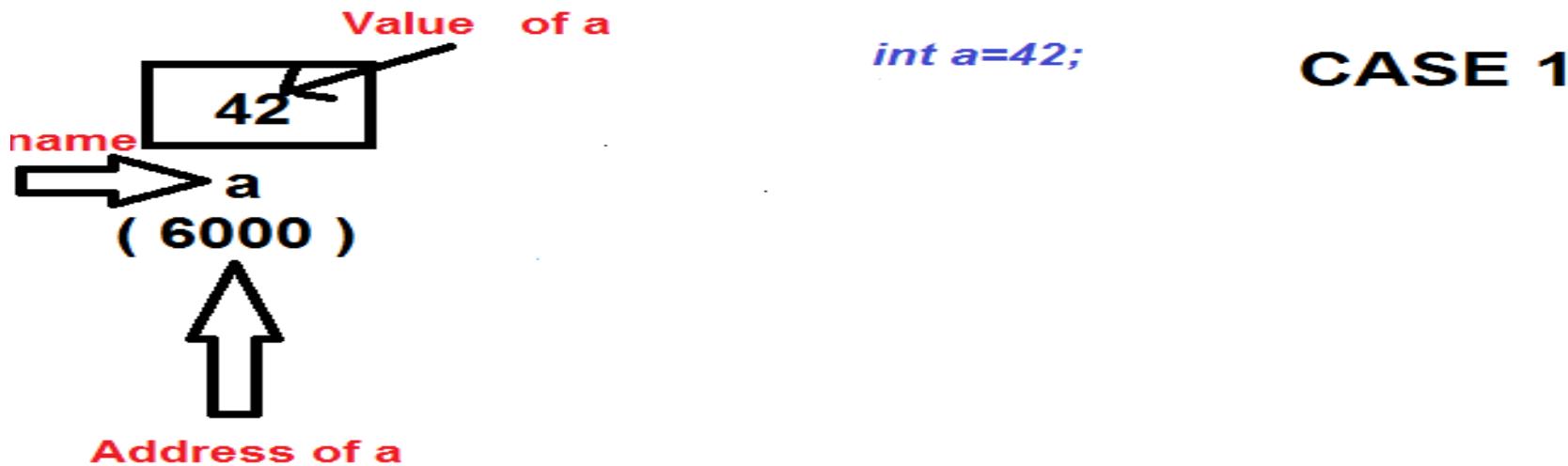


- A **variable** can be seen as a **container** to store certain values.
- While the program is running, **variables** are **accessed** and sometimes **changed**, i.e. a **new value** will be **assigned** to a **variable**

How Variables Work In C ?



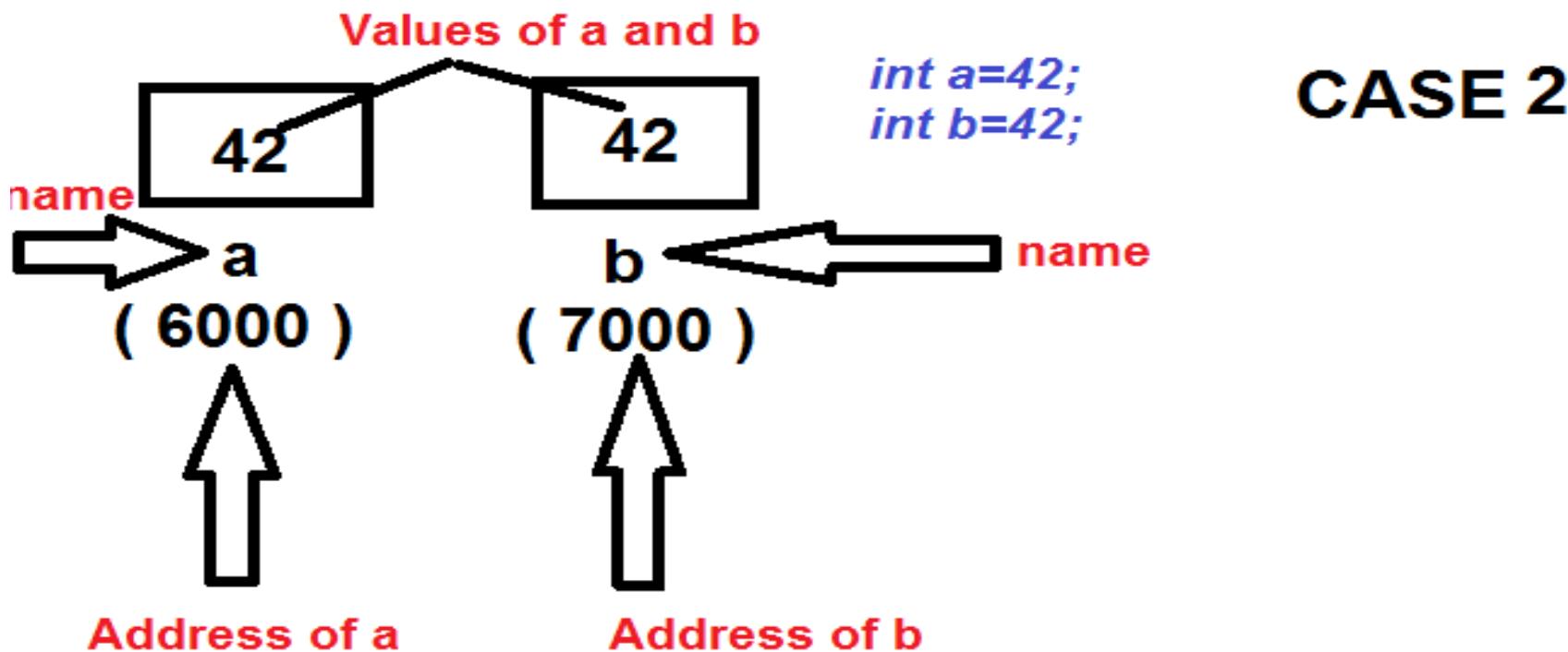
- In **C language** when we declare a **variable** and **assign** value to it then **some space is created** in memory by the **given name** and the **given value** is stored in it.
- Suppose we write the statement **int a=42;** , then the following will be the memory diagram.



How Variables Work In C ?



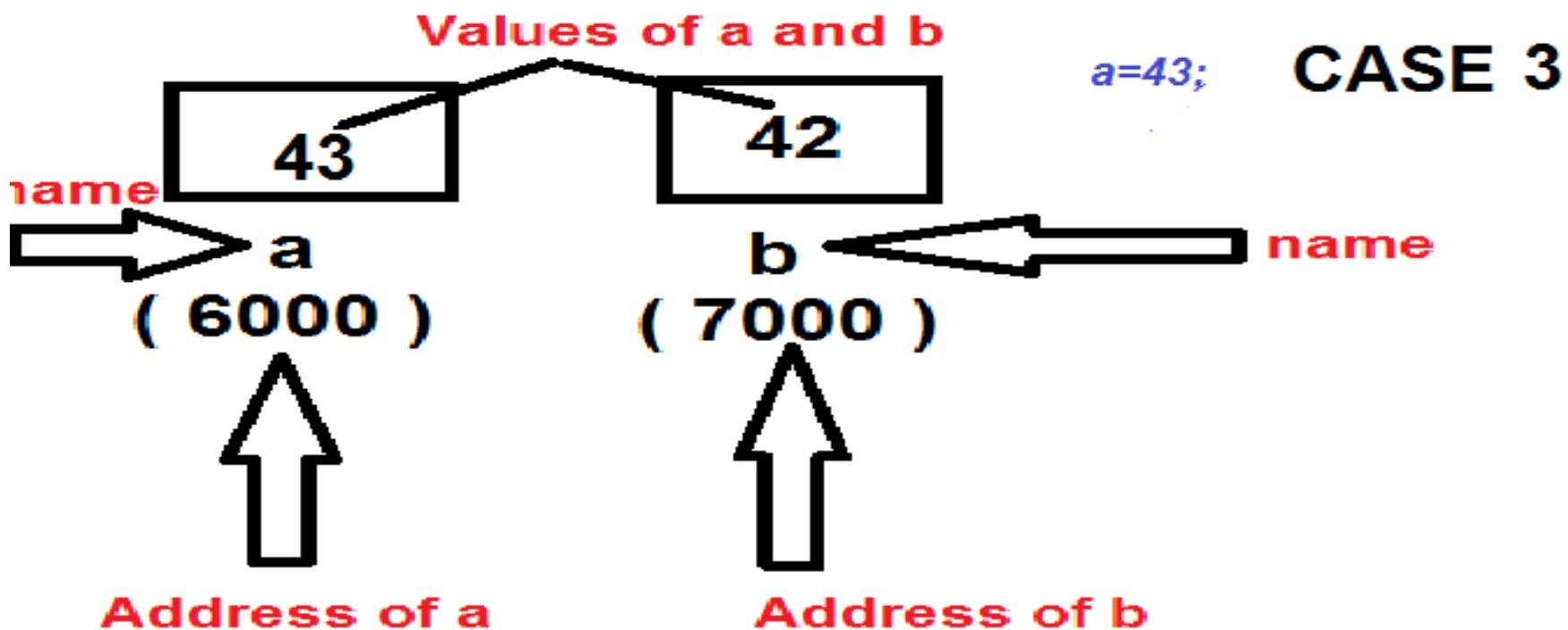
- Now if we declare **another variable**, with the **same value**, then again the **same process** will take place.
- Suppose we write , **int b=42;**



How Variables Work In C ?



- Finally if we **assign** a **new value** to an **existing variable** , then it's previous value gets **overwritten**
- Suppose we write , **a=43;**



How Variables Work In Python ?

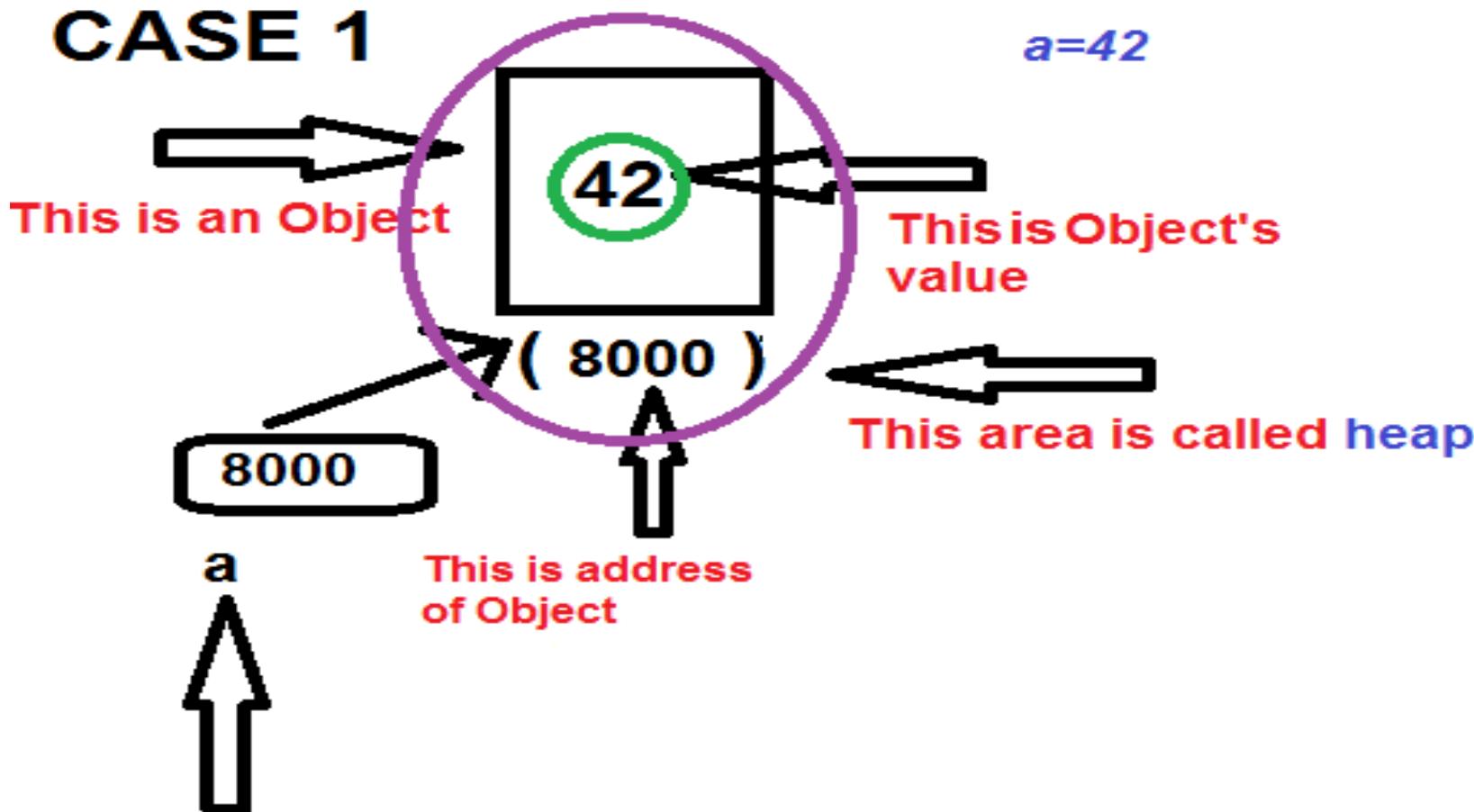


- In **Python** when we **assign** value to a **variable** , then things are **different** than **C** .
- Suppose we write **a=42** in **Python** , then **Python** will create 2 things:
 - An **object** in **heap memory** holding the **value 42** , and
 - A **reference** called **a** which will point to this **object**

How Variables Work In Python ?



CASE 1



a is called reference or tag , and it holds the address of the object

How Variables Work In Python ?

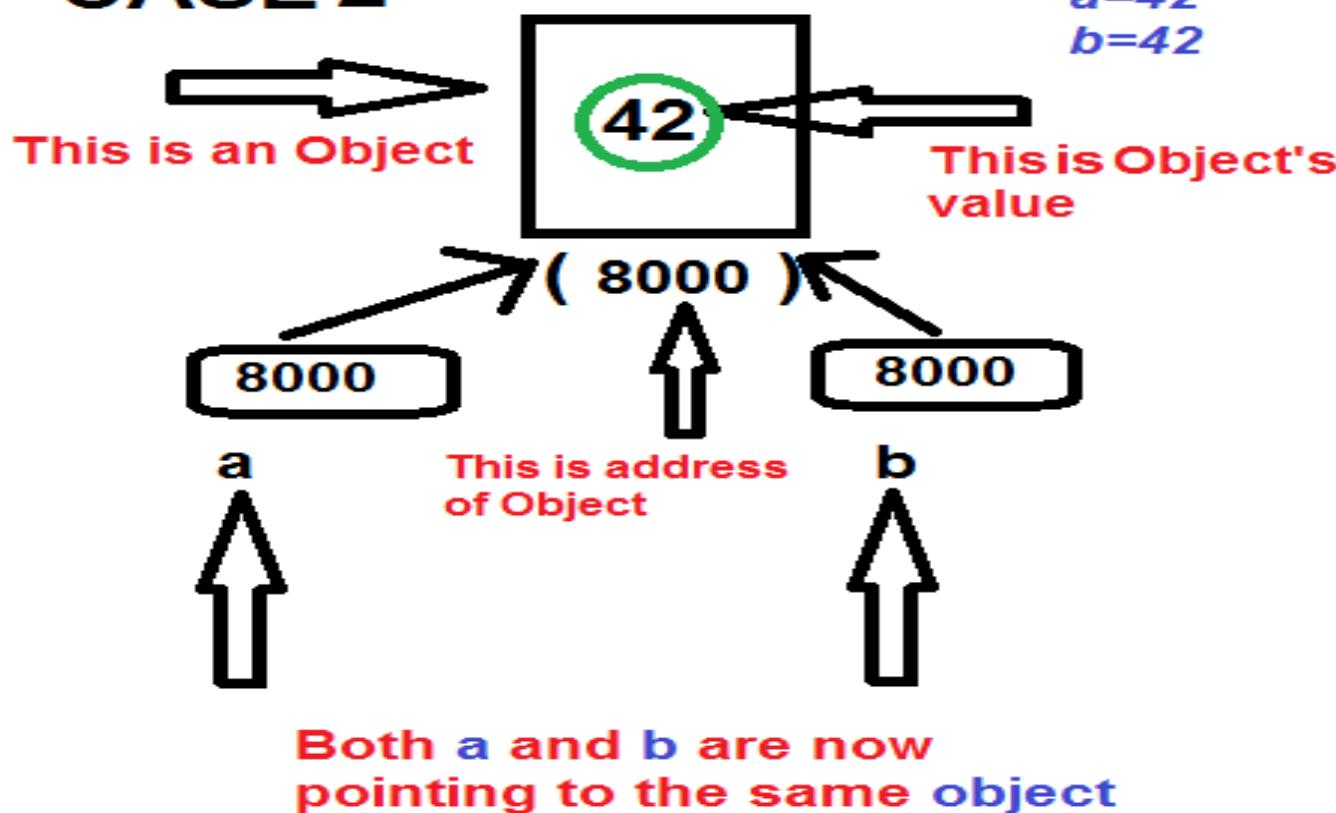


- Now if we **create another variable** called **b** and **assign** it the **same value**, then **Python** will do the following:
 - Create a new reference by the name b**
 - Assign the address of the previously created object to the reference b because the value is same**
 - So now both a and b are pointing to the same object**

How Variables Work In Python ?



CASE 2



How Variables Work In Python ?

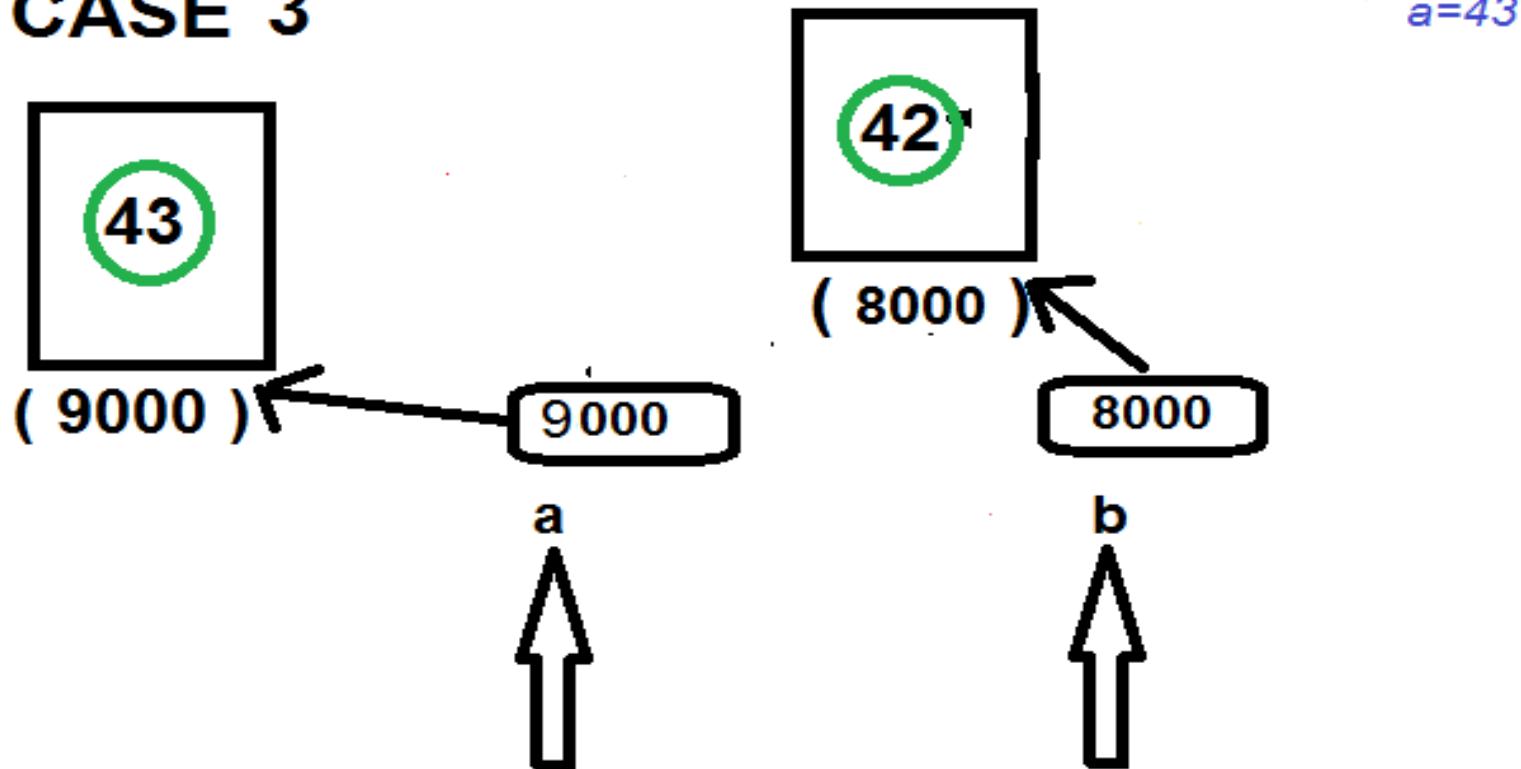


- Finally if we **assign** a new value to the variable **a** , then like **C** , **Python** will **not overwrite** the **value** .
- Rather it will do the following:
 - **Create a new object initialized with the new value**
 - **Assign the address of the newly created object to the reference a**
 - **However the reference b is still pointing to the same object**

How Variables Work In Python ?



CASE 3



The reference `a` is now pointing to a new object

How Variables Work In Python ?



- This behaviour of **objects** in **Python** is called **“immutability”**
- In other words when we cannot change the value of an **object** , we say it is **immutable** , otherwise we say it is **mutable**
- Objects of **built-in types** like (**int**, **float**, **bool**, **str**, **complex**, **tuple**) are **immutable**.
- Objects of **built-in types** like (**list**, **set**, **dict**) are **mutable**.



Strings Are Also Immutable

- **String** objects in **Python** are also **immutable**.
- That is , once we have created a **String** object , then we **cannot overwrite** it's value.
- Although we can **change** the **value** of the **String** reference by **assigning** it new **String**.



Strings Are Also Immutable



```
>>> city="Bhopal"
>>> print(city)
Bhopal
>>> city[0]='c'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> city="Indore"
>>> print(city)
Indore
```



Immutable And Mutable

- Following data types in **Python** are **immutable**:
 - **int**
 - **float**
 - **bool**
 - **str**
 - **tuple**
 - **complex**
 - **range**
 - **frozenset**
 - **bytes**
- Following data types in **Python** are **mutable**:
 - **list**
 - **dict**
 - **set**
 - **bytearray**

These Notes Have Python - world _ In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

Two Very Important Questions



- Keeping in mind the concept of immutability , **2 very common questions** arise :
- **What is the benefit of making objects immutable ?**
&
- **If we have a single reference and we create multiple objects , then wouldn't there be memory wastage ?**



Qn 1 : Benefit Of Immutability

- The **main benefit** of **immutability** is that , it prevents unnecessary creation of new objects .
- This is because if we write , the following **2 statements**: **a=42**
b=42
- Then **Python** will not create **2 objects** . Rather it only creates **1 object** and makes both the references **a** and **b** ,**refer** to the **same object**.
- This **saves memory** and **overhead** of creating **multiple objects**

Qn 2 : What About Single Reference And Multiple Objects



- Consider the following **3 lines**:

a=10

a=20

a=30

- When the above **3 lines** will run , then **Python** will create **3 objects** , **one by one** and finally the **reference a** will **refer** to the last object with the value **30**
- An obvious question arises , that **what will happen** to the previous **2 objects** with the value **10** and **20** ?



Three Important Terms

- Before understanding, what will happen to the previous 2 objects , we need to understand **3 important terminologies:**
 1. Garbage Block
 2. The Garbage Collection
 3. Reference Counting

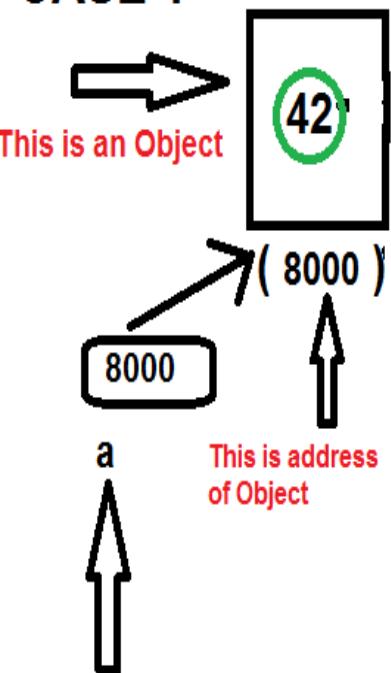


The Garbage Block

- In **Python**, if an **object** is **not being referred** by any **reference**, then such **objects** are called **Garbage Blocks**.

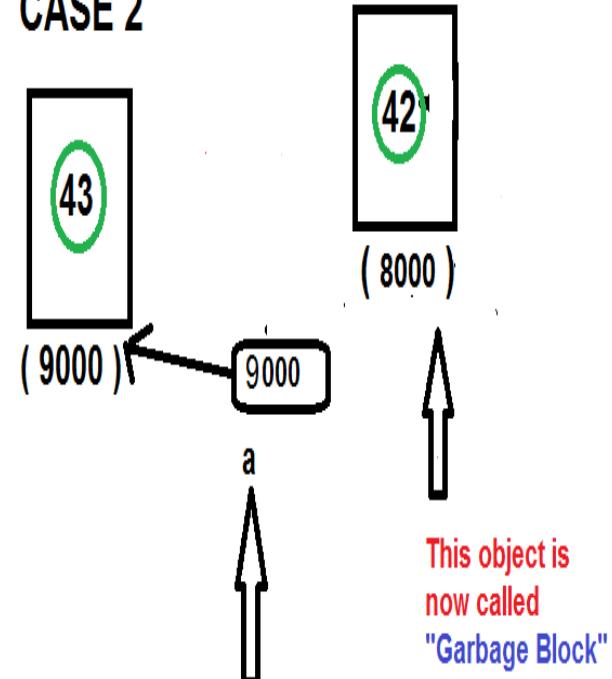
The Garbage Block

CASE 1



`a=42`

CASE 2



`a` is the reference holding the object's address

The reference `a` is now pointing to a new object



The Garbage Collection

- **Garbage collection** is the process of **cleaning the computer memory** which was used by a **running program** when that program **no longer needs that memory** .
- **Garbage collection** is a **memory management** feature in **many** programming languages.



Reference Counting

- The process of **memory management** in **Python** is **straightforward**.
- **Python** handles it's **objects** by keeping a **count** of the **number of references** each **object** has in the program.
- In **simple words** it means, each **object** stores how many **references** are **currently referring** it.



Reference Counting

- This **count** is **updated** with the **program runtime** and when it reaches **0**, this means it is not **reachable** from the **program** anymore.
- Hence, the **memory** for this **object** can be **reclaimed** and be **freed** by the **interpreter**.



Reference Counting

```
a=10 // The object 10 has a reference count of 1  
b=10 // Now 10 has reference count of 2  
a=20 // Now reference count of 10 becomes 1  
b=20 // Finally reference count of 10 becomes 0
```

- As soon as the reference count of 10 becomes 0 , Python automatically removes the object 10 from memory



Reference Counting

- So in our example

a=10

a=20

a=30

- The objects **10** and **20** will be **reclaimed** by the **Python Garbage Collector** as soon as their **reference count** becomes **0**



Reference Counting

- **ADVANTAGE:**

- The **main advantage** of such approach is that **unused memory** is **reclaimed** and **made available** for **use** again.

- **DISADVANTAGE:**

- The **drawback** is that **Python** has to **continuously watch** the **reference count** of **objects** in the **background** and **free** them **as soon as** the **reference count** becomes **0**.
- This is **another important reason** why **Python** is **slow** as **compared to** other languages



Another Important Question

- Consider the following statements:

a=42

b=42

- Can you tell how many objects has Python created in the above code ?
- Answer: Only 1
- But, what is the proof?



Another Important Question

- We can prove this in 2 ways:
 - By using **id()** function
 - By using **is** operator

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



The id() Function

- **id()** is a built-in function in **Python 3**, which returns the ***identity*** of an **object**.
- The ***identity*** is a **unique integer** for that **object** during its lifetime.
- This is also the **address** of the **object** in **memory**.



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 9



Today's Agenda

- **Comments , Constants And More About print() Function**
 - **How to write Comments in Python ?**
 - **How to create constants in Python ?**
 - **How to print a variables value using print() ?**



Comments In Python

- **Comments** are statements in our program which are ignored by the **compiler** or **interpreter** i.e *they are not executed by the language*
- We generally create comments to **let developers understand** our **code's logic**.
- This is a **necessary practice**, and **good developers** make heavy use of the **comment system**.
- Without it, things can get confusing

Types Of Comments In Python



- Python provides **2** types of **comments**:
 - **Single Line Comment (official way of comment)**
 - **MultiLine Comment (un official way)**



Single Line Comments

- **Single-line comments** are created simply by beginning a line with the **hash (#)** character, and they are automatically terminated by the end of line.
- **For example:**

```
a=10  
#a=a+1  
print(a)
```

Output:

10



This line gets commented out and is not executed

Official Way Of Multi Line Comments



- To create a **Multi Line Comments**, the only problem with this style is we will have prefix each line with **#**, as shown below:

```
#a=a+1
```

```
#b=b+5
```

```
#c=c+10
```

- But most **Python** projects follow this style and **Python's PEP 8 style guide** also **favours** repeated **single-line comments**.



What Is PEP ?

- **PEP** stands for **Python Enhancement Proposal**.
- It is **Python's style guide** and is officially called **PEP8**
- In simple words it is a set of rules for how to format your **Python code** to **maximize its readability** .
- We can find it at <https://www.python.org/dev/peps/pep-0008/>



Why Is PEP Needed ?

- When you **develop a program** in a **group of programmers**, it is **really important** to follow **some standards**.
- If all **team members** format the code in the **same prescribed format**, then it is **much easier** to **read** the code.
- For the same purpose **PEP8** is used to ensure **Python** coding standards are met.

Un Official Way Of Multi Line Comments



- If we want to simplify our efforts for writing **Multi Line Comments**, then we can wrap these comments inside **triple quotes** (double or single) as shown below
- For example:

```
a=10
'''a=a+1
a=a+1 '''
print(a)
```

Output:

10



Both the lines
get
commented
out and are
not executed



Why It Is UnOfficial ?

- **Triple quotes doesn't create “true” comments.**
- They are **regular multiline strings**, but since they are not getting assigned to any **variable**, they will get **garbage collected** as soon as the code runs.
- Hence they are **not ignored by the interpreter** in the same way that **#a** comment is.



What Is A Constant ?



- A **constant** is a type of variable whose value cannot be changed.
- **C++** provides the keyword **const** for declaring constant as shown below:

```
const float pi=3.14;  
pi=5.0; // Syntax Error
```
- **Java** provides the keyword **final** for declaring constant:

```
final double PI=3.14;  
PI=5.0; // Syntax Error
```

How To Create A Constant In Python?



- Unfortunately, there is no **keyword** in **Python**, like **const** or **final**, to declare a **variable** as **constant**.
- This is because of **dynamic nature** of **Python** .
- However there is a **convention** in **Python**, that we can follow to let other developer's know that we are declaring a **variable** as **constant** and we don't want others to change it's value.
- The convention is to declare the variable in all **upper case**

How To Create A Constant In Python?



- For example:

`PI=3.14`

`MAX_MARKS=100`

- But again , remember this is just a convention not a rule and still the value of `PI` and `MAX_MARKS` can be changed

Some More About print() Function



- We know that **print()** function can be used to print **messages** on the **output screen**.
- But we can also use **print()** to display **single** or **multiple** variable values.
- We just have to separate them with comma :
 - **print(arg1 , arg2, arg3, ...)**



The print() Function

- Example:
a="Good"
b="Morning"
print(a+b)
- Output:
GoodMorning
- Example:
a="Good"
b=10
print(a+b)
- Output:
TypeError



The print() Function

- **Example:**

```
a="Good"  
b="Morning"  
print(a,b)
```

- **Output:**

Good Morning

- **Example**

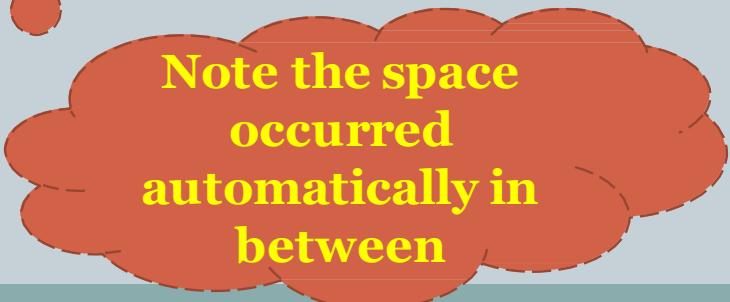
```
:
```



```
a="Good"  
b=10  
print(a,b)
```

- **Output:**

Good 10



Note the space
occurred
automatically in
between



The print() Function

- Example

e: a=10

b=20

print(a,b)

- Example

:

a="Good"

b=10

print(a,b)

- Output:

10 20

- Output:

Good 10



The print() Function

- **Example:**

```
name="Sachin"
```

```
print("My name is", name)
```

- **Output:**

My name is Sachin



The print() Function

- **Example:**

age=32

```
print("My age is", age)
```

- **Output:**

My age is 32



The print() Function

- **Example:**

```
name="Sachin"
```

```
age=32
```

```
print("My name is",name,"and my age is",age)
```



Note , we have not provided any space at marked positions but in the output we will automatically get the space

- **Output:**

My name is Sachin and my age is 32

How Is Space Getting Generated?



- Just like `print()` function has a **keyword argument** called `end` , which generates `newline` automatically , similarly it also has another keyword argument called `sep`
- This argument has the default value of `" "` and is used by **Python** to separate values of **2 arguments** on screen.

How Is Space Getting Generated?



- So the statement:
 - `print("Good", "Morning")`
- Is actually converted by **Python** to
 - `print("Good", "Morning", sep=" ")`
- And the output becomes
 - **Good Morning**

Changing The Default Value Of sep



- We can change the default value of **sep** to any value we like
 -
- To do this , we just have to pass sep as the last argument to the function **print()**
 - `print("Good","Morning",sep=",")`
- And the output becomes
 - **Good,Morning**

Note that comma has occurred instead of space



The print() Function

- Example:

le: a=10

b=20

print(a,b,sep="#")

- Output:

10#20

- Example:

le:

hh=10

mm=30

ss=45

print(hh,mm,ss,sep=":")

- Output:

10:30:45



These Notes Have Python _ world In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 10



QUIZ 2- Test Your Skills



1. What is the maximum possible length of an identifier in Python?

- A. 31 characters
- B. 63 characters
- C. 79 characters
- D. none of the mentioned

Correct Answer: D



QUIZ 2- Test Your Skills



2. Which of these is not a core data type in Python?

- A. Class
- B. List
- C. Str
- D. Tuple

Correct Answer: A



QUIZ 2- Test Your Skills



3. Following set of commands are executed in shell, what will be the output?

```
>>>str="hello"
```

```
>>>str[:2]
```

- A. hel
- B. he
- C. Lo
- D. olleh

Correct Answer: B



QUIZ- Test Your Skills



4. What is the return type of function id ?

- A. int
- B. float
- C. bool
- D. dict

Correct Answer: A



QUIZ- Test Your Skills

5. Which of the following results in a SyntaxError ?

- A. ' "Once upon a time...", she said. '
- B. "He said, 'Yes!' "
- C. '3\'
- D. "That's okay"

Correct Answer: C



QUIZ- Test Your Skills



6. Which of the following is not a complex number?

- A. $k = 2 + 3j$
- B. $k = \text{complex}(2)$
- C. $k = 2 + 3l$
- D. $k = 2 + 3J$

Correct Answer: C



QUIZ- Test Your Skills



7. Which of the following is incorrect?

- A. k = 0b101
- B. k= 0x4f5
- C. k = 19023
- D. k = 0o3964

Correct Answer: D



QUIZ- Test Your Skills



8. What is the output of the code:

```
print(bool('False'))
```

- A. False
- B. True
- C. SyntaxError
- D. o

Correct Answer: B



QUIZ- Test Your Skills



9. Out of List and Tuple which is mutable ?

- A. List
- B. Tuple
- C. Both
- D. None

Correct Answer: A



QUIZ- Test Your Skills



10. Are string references mutable ?

- A. Yes
- B. No

Correct Answer: A



QUIZ- Test Your Skills



11. Are string objects mutable ?

- A. Yes
- B. No

Correct Answer: B



QUIZ- Test Your Skills



12. Is there a do – while loop in Python ?

- A. Yes
- B. No

Correct Answer:B



QUIZ- Test Your Skills

13. In Python which is the correct method to load a module ?

- A. include math
- B. import math
- C. #include<math.h>
- D. using math

Correct Answer: B



QUIZ- Test Your Skills



14. What is the name of data type for character in Python ?

- A. chr
- B. char
- C. str
- D. None Of The Above

Correct Answer: D



QUIZ- Test Your Skills



15. Let a = "12345" then which of the following is correct ?

- A. print(a[:]) will show 1234
- B. print(a[0:]) will show 2345
- C. print(a[:100]) will show 12345
- D. print(a[1:]) will show 1

Correct Answer: C



Today's Agenda

- Operators In Python
 - Types Of Operators
 - Arithmetic Operators
 - Special points about + and *
 - Difference between / and //



Operators

- Operators are special symbols in that carry out **different kinds** of **computation** on values.
- For example : **2+3**
- In the expression **2+3** , **+** is an operator which performs **addition** of **2** and **3** , which are called operands

Types Of Operators In Python



- Python provides us **6** popular types of **operators**:
 - **Arithmetic Operators**
 - **Relational or Comparison Operators**
 - **Logical Operators**
 - **Assignment Operator**
 - **Identity Operators**
 - **Membership Operators**

Arithmetic Operators In Python



- In **Python**, we have **7 arithmetic operators** and they are as below:

+

(Arithmetic Addition)

-

(Subtraction)

*

(Arithmetic Multiplication)

/

(Float Division)

%

(Modulo Division)

//

(Floor Division)

**

(Power or Exponentiation)

The 5 Basic Arithmetic Operators



mymath.py

```
a=10
```

```
b=4
```

```
print("sum of",a,"and",b,"is",a+b)
```

```
print("diff of",a,"and",b,"is",a-b)
```

```
print("prod of",a,"and",b,"is",a*b)
```

```
print("div of",a,"and",b,"is",a/b)
```

```
print("rem of",a,"and",b,"is",a%b)
```

The 5 Basic Arithmetic Operators



The Output:

```
D:\My Python Codes>python mymath.py
sum of 10 and 4 is 14
diff of 10 and 4 is 6
prod of 10 and 4 is 40
div of 10 and 4 is 2.5
rem of 10 and 4 is 2
```

Two Special Operators // and **



- The operator **//** in **Python** is called as **floor division**.
- Means it **returns** the **integer part** and not the **decimal part**.
- For example: **5//2** will be **2** not **2.5**

Two Special Operators // and **



- But there are 3 very important points to understand about this operator
- When used with **positive numbers** the result is **only the integer part** of the **actual answer** i.e., **the decimal part is truncated**
- However if one of the **operands is negative**, the result is **floored**.
- If **both** the **operands** are **integers**, result will also be **integer**, otherwise result will be **float**



The Floor Division Operator

- **Example:**

a=10

b=4

print(a//b)

- **Example:**

a=10.0

b=4

print(a//b)

- **Output:**

2

- **Output:**

2.0

If both the operands are integers , the result is also an integer . But if any of the operands is float the result is also float



The Floor Division Operator

- Example:
a=97
b=10
print(a//b)
- Output:
9
- Example:
a=97
b=10.0
print(a//b)
- Output:
9.0



The Floor Division Operator

- Example:
 `a=-10` `b=4`
 `print(a//b)`
- Output:
 `-3`
- Example:
 `a=19` `b=-2`
 `print(a//b)`
- Output:
 `-10`



The Floor Division Operator

- Example:
ie: a=-
 b=-4
`print(a//b)`
- Output:
`9`
- Example:
ie: a=-
 b=-2
`print(a//b)`



An Important Point

- There is another **very important point** to remember about the **3** operators **/**, **//** and **%**
- The point is that if the **denominator** in these **operators** is **0** or **0.0**, then **Python** will throw the exception called **ZeroDivisionError**



Division By 0

- Example:

e: `a=10`

`b=0`

`print(a/b)`

- Example:

`a=10`

`b=0.0`

`print(a/b)`

- Output:

`ZeroDivisionError`

- Output:

`ZeroDivisionError`



Division By 0

- Example:

ie:

a=10

b=0

print(a//b)

- Example:

a=10

b=0.0

print(a//b)

- Output:

ZeroDivisionError

- Output:

ZeroDivisionError



Division By 0

- Example:

ie:

a=10

b=0

print(a%b)

- Example:

a=10

b=0.0

print(a%b)

- Output:

ZeroDivisionError

- Output:

ZeroDivisionError



The power (**)-Operator

- The **power operator** i.e. ****** performs **exponential (power)** calculation on operands.

- **For example:**

```
a=10
```

```
b=3
```

```
print(a**b)
```

- **Output:**

1000



Double Role Of The Operator +

- The operator **+** as discussed earlier also ,has **2 roles** in **Python**
- When used with **numbers** , it performs **addition** and when used with **strings** it performs **concatenation**
- For example:

a=10

b=5

Output: print(a+b)

15

a="Good"

b="Evening"

print(a+b)

Output:
GoodEvening



Double Role Of The Operator +

- Example:

a="Good"

b=10

print(a+b)

- Example:

a="Good"

b="10"

print(a+b)

- Output:

TypeError

- Output:

Good10



Double Role Of The Operator *

- The operator * also has **2 roles** in Python
- When used with **numbers**, it performs **multiplication** and when used with **one operand string** and **other operand int** it performs **repetition**
- For

example:

a=10

b=5

Output: print(a*b)

50

a="Sachin"

b=3

print(a*b)

Output:

SachinSachinSachin



The * Operator

- Example:
 ie: a=5
 b=4.0
 print(a*b)
- Output:
 20.0
- Example:
 a="Sachin"
 b=3.0
 print(a*b)
- Output:
 Type Error :
 Can't multiply
 by non int



The * Operator

- Example:
a="Sachin"
b=3
print(b*a)
- Output:
SachinSachinSachin
- Example:
a="Sachin"
b="Kapoor"
print(a*b)
- Output:
**Type Error :
Can't multiply
by non int**



PYTHON
LECTURE 11



Today's Agenda



- Operators In Python
 - Relational Operators
 - Relational Operators With Strings
 - Chaining Of Relational Operators
 - Special Behavior Of == and !=



Relational Operators In Python



- **Relational operators** are used to **compare** values.
- They either return **True** or **False** according to the condition.
- These operators are:

Operator	Meaning
>	Greater Than
<	Less Than
>=	Greater Than Equal To
<=	Less Than Equal To
==	Equal To
!=	Not Equal To



The 6 Basic Relational Operators



myrelop.py

```
a=10
```

```
b=4
```

```
print("a=",a,"b=",b)
```

```
print("a > b",a>b)
```

```
print("a < b",a<b)
```

```
print("a==b",a==b)
```

```
print("a!=b",a!=b)
```

```
print("a>=b",a>=b)
```

```
print("a<=b",a<=b)
```



The 6 Basic Relational Operators



The Output:

```
D:\My Python Codes>python myrelop.py
a= 10 b= 4
a > b True
a < b False
a==b False
a!=b True
a>=b True
a<=b False
```



Relational Operators With Strings



- Relational Operators can also work with strings .
- When applied on string operands , they compare the unicode of corresponding characters and return True or False based on that comparison.
- As discussed previously , this type of comparison is called lexicographical comparsion



Relational Operators With Strings



myrelop2.py

```
a="Ramesh"  
b="Rajesh"  
print("a=",a,"b=",b)  
print("a > b",a>b)  
print("a < b",a<b)  
print("a==b",a==b)  
print("a!=b",a!=b)  
print("a>=b",a>=b)  
print("a<=b",a<=b)
```



Relational Operators With Strings



The Output:

```
D:\My Python Codes>python myrelop2.py
a= Ramesh b= Rajesh
a > b True
a < b False
a==b False
a!=b True
a>=b True
a<=b False
```



Relational Operators With Strings



- If we want to check the **UNICODE** value for a **particular letter**, then we can call the function **ord()**.

- It is a **built in function** which accepts **only one character** as argument and it returns the **UNICODE** number of the **argument passed**

□ Example:

ord('A')

65

ord('m')

109

ord('j')

106



Relational Operators With Strings

myrelop4.py

```
a= "BHOPAL"  
b= "bhopal"  
print("a=",a,"b=",b)  
print("a > b",a>b)  
print("a < b",a<b)  
print("a==b",a==b)  
print("a!=b",a!=b)  
print("a>=b",a>=b)  
print("a<=b",a<=b)
```



Relational Operators With Strings



The Output:

```
D:\My Python Codes>python myrelOp4.py
a= BHOPAL b= bhopal
a > b False
a < b True
a==b False
a!=b True
a>=b False
a<=b True
```



Will This Code Run ?

```
a=True  
b=False  
  
print("a=",a,"b=",b)  
print("a > b",a>b)  
print("a < b",a<b)  
print("a==b",a==b)  
print("a!=b",a!=b)  
print("a>=b",a>=b)  
print("a<=b",a<=b)
```

Yes , the code will successfully
Run because True is 1 and False is 0

Output:

```
D:\My Python Codes>python myrellop5.py  
a= True b= False  
a > b True  
a < b False  
a==b False  
a!=b True  
a>=b True  
a<=b False
```



What about this code?

```
a='True'  
b='False'  
print("a=",a,"b=",b)  
print("a > b",a>b)  
print("a < b",a<b)  
print("a==b",a==b)  
print("a!=b",a!=b)  
print("a>=b",a>=b)  
print("a<=b",a<=b)
```

Yes , this code will also successfully Run but 'True' and 'False' will be handled as strings

Output:

```
D:\My Python Codes>python myrellop6.py  
a= True b= False  
a > b True  
a < b False  
a==b False  
a!=b True  
a>=b True  
a<=b False
```

Special Behavior Of Relational Operators



- Python allows us to **chain** multiple **relational operators** in one **single statement**.
- For example the expression **1<2<3** is **perfectly valid** in **Python**
- However when **Python** evaluates the expression , it returns **True if all individual conditions are true** , otherwise it returns **False**



Cascading Of Relational Operators

- **Example:**
`print(7>6>5)`
- **Output:**
True
- **Example:**
`print(5<6>7)`
- **Output:**
False



Cascading Of Relational Operators

- **Example:**
`print(5>6>7)`
- **Output:**
`False`
- **Example:**
`print(5<6<7)`
- **Output:**
`True`

Special Behavior Of

`== And !=`



- `==` compares its **operands** for **equality** and if they are of **compatible types** and **have same value** then it returns **True** otherwise it returns **False**

- Similarly `!=` compares its **operands** for **inequality** and if they are of **incompatible types** or **have different value** then it returns **True** otherwise it returns **False**

Special Behavior Of

`== And !=`



- Example:

```
print(10==10)
```

- Output:

True

- Example:

```
print(10==20)
```

- Output:

False

Special Behavior Of

`== And !=`



- Example:

```
print(10=="10")
```

- Output:

False

- Example:
`print(10==True)`

- Output:
False

Special Behavior Of

== And !=



- **Example:**

```
print(1==True)
```

- **Output:**

True

- **Example:**
`print("A"=="A")`

- **Output:**
True

Special Behavior Of

`== And !=`



- Example:

```
print("A"=="65")
```

- Output:

False

- Example:
`print("A"==65)`

- Output:
False

Special Behavior Of

`== And !=`



- Example:

```
print(15==15.0)
```

- Example:
`print(15==15.01)`

- Output:

True

- Output:
False

Special Behavior Of

`== And !=`



- Example:

```
print(15!="15")
```

- Output:

True

- Example:
`print(o != False)`

- Output:
`False`

Special Behavior Of

`== And !=`



- Example:

```
print(False!=True)
```

- Output:

True

- Example:

```
print(False != 0.0)
```

- Output:

False

Special Behavior Of

`== And !=`



- Example:

```
print(2+5j==2+5j)
```

- Output:

True

- Example:
`print(2+5j!= 2)`

- Output:
True



PYTHON
LECTURE 12



Today's Agenda

- Operators In Python
 - Logical Operators
 - How Logical Operators Work With Boolean Types ?
 - How Logical Operators Work With Non Boolean Types ?



Logical Operators In Python



- **Logical operators** are used to combine **two or more conditions** and perform the logical operations using **Logical and**, **Logical or** and **Logical not**.

Operator	Meaning
and	It will return true when both conditions are true
or	It will returns true when at-least one of the condition is true
not	If the condition is true, logical NOT operator makes it false



Behavior Of Logical **and** Operator



```
>>> a=40  
>>> b=20  
>>> c=50  
>>> a>b and a>c
```

```
False
```

```
>>> a=40  
>>> b=20  
>>> c=50  
>>> a>b and c>a
```

```
True
```



Behavior Of Logical or Operator



```
>>> a=40  
>>> b=20  
>>> c=50  
>>> a>b or a>c
```

True

```
>>> a=40  
>>> b=20  
>>> c=50  
>>> b>a or b>c
```

False



Behavior Of Logical **not** Operator



```
>>> a=True  
>>> not a  
False  
>>> b=False  
>>> not b  
True
```

Behavior Of Logical Operators With Non Boolean



- Python allows us to apply logical operators with non boolean types also
- But before we understand how these operators work with non boolean types, we must understand some very important points

Behavior Of Logical Operators With Non Boolean



1. **None, 0 , 0.0 , ""** are all **False** values

1. The return value of **Logical and & Logical or operators** is never **True** or **False** when they are applied on **non boolean** types.

Behavior Of Logical Operators With Non Boolean



3. If the **first value** is **False** , then **Logical and** returns **first value** , otherwise it returns the **second value**

3. If the **first value** is **True** , then **Logical or** returns **first value** , otherwise it returns the **second value**

3. When we use **not operator** on **non boolean** types , it returns **True** if it's operand is **False**(in any form) and **False** if it's operand is **True** (in any form)

Logical Operators On Non Boolean Types



- Example:
5 and 6
- Output:
6
- Example:
5 and 0
- Output:
0

Logical Operators On Non Boolean Types



- **Example:**
0 and 10
- **Output:**
0
- **Example:**
6 and 0
- **Output:**
0

Logical Operators On Non Boolean Types



- Example:
'Sachin' and 10
- Output:
10
- Example:
'Sachin' and 0
- Output:
0

Logical Operators On Non Boolean Types



- Example:
'Indore' and 'Bhopal'
- Output:
Bhopal
- Example:
'Bhopal' and 'Indore'
- Output:
Indore

Logical Operators On Non Boolean Types



- Example:
`0 and 10/0`
- Output:
`0`
- Example:
`10/0 and 0`
- Output:
`ZeroDivisionError`

Logical Operators On Non Boolean Types



- Example:

5 or 6

- Output:

5

- Example:

5 or 0

- Output:

5

Logical Operators On Non Boolean Types



- Example:
0 or 10
- Output:
10
- Example:
6 or 0
- Output:
6

Logical Operators On Non Boolean Types



- Example:
'Sachin' or 10
- Output:
Sachin
- Example:
'Sachin' or 0
- Output:
Sachin

Logical Operators On Non Boolean Types



- Example:
'Indore' or 'Bhopal'
- Output:
Indore
- Example:
'Bhopal' or 'Indore'
- Output:
Bhopal

Logical Operators On Non Boolean Types



- Example:
0 or 10/0
- Output:
ZeroDivisionError
- Example:
10/0 or 0
- Output:
ZeroDivisionError

Logical Operators On Non Boolean Types



- **Example:**
not 5
- **Output:**
False
- **Example:**
not 0
- **Output:**
True

Logical Operators On Non Boolean Types



- **Example:**
`not 'Sachin'`
- **Output:**
`False`
- **Example:**
`not ''`
- **Output:**
`True`



PYTHON
LECTURE 13



Today's Agenda

- **Operators In Python**
 - **Assignment Operators**
 - **Various Types Of Assignment Operators**
 - **Compound Operators**
 - **Identity Operators**
 - **Membership Operators**
 - **Precedence And Associativity**



Assignment Operators In Python



- The **Python Assignment Operators** are used to **assign** the values to the **declared variables**.

- **Equals (=) operator** is the **most commonly used assignment operator** in Python.

- **For example:**
 - **a=10**



Assignment Operators In Python



□ Shortcut for assigning same value to all the variables

□ **x=y=z=10**

□ Shortcut for assigning different value to all the variables

□ **x,y,z=10,20,30**



Guess The Output



```
a,b,c=10,20  
print(a,b,c)
```

Output:

ValueError : Not enough values to unpack

```
a,b,c=10,20,30,40  
print(a,b,c)
```

Output:

ValueError : Too many values to unpack



Compound Assignment Operators



- Python allows us to combine **arithmetic operators** as with **assignment operator**.

- For example: The statement
 - **x=x+5**

- Can also be written as
 - **x+=5**



Compound Assignment Operators



Operator	Example	Meaning
<code>+=</code>	<code>x+=5</code>	<code>x=x+5</code>
<code>-=</code>	<code>x-=5</code>	<code>x=x-5</code>
<code>*=</code>	<code>x*=5</code>	<code>x=x*5</code>
<code>/=</code>	<code>x/=5</code>	<code>x=x/5</code>
<code>%=</code>	<code>x%=5</code>	<code>x=x%5</code>
<code>//=</code>	<code>x//=5</code>	<code>x=x//5</code>
<code>**=</code>	<code>x**=5</code>	<code>x=x**5</code>



Guess The Output



a=10

print(++a)

Output:

10

a=10

print(a++)

Output:

SyntaxError : Invalid Syntax

Conclusion:

Python does not have any
increment operator like `++`.

Rather it is solved as

`+(+x)` i.e `+(+10)` which is **10**

However the expression `a++`
is an error as it doesn't make
any sense



Guess The Output



a=10

print(--a)

Output:

10

a=10

print(a--)

Output:

SyntaxError : Invalid Syntax

Conclusion:

Python does not have any **decrement operator** like --.

Rather it is solved as

-(-x) i.e -(-10) which is 10

However the expression a-- is an error as it doesn't make any sense



Guess The Output



```
a=10  
print(+++++a)
```

Try to figure out yourself
the reason for these outputs

Output:

10

```
a=10  
print(-----a)
```

Output:

-10



Identity Operators



- Identity operators in Python are **is** and **is not**
- They serve 2 purposes:
 - To verify if **two references point to the same memory location or not**
 - To determine **whether a value is of a certain class or type**

AND



Behavior Of **is** and **is not**



- The operator **is** returns **True** if the operands are **identical** , **otherwise** it returns **False**.

- The operator **is not** returns **True** if the operands are **not identical** , **otherwise** it returns **False**.



Examples Of **is** Operator



a=2

b=3

c=a **is** b

print(c)

Output:

False

Explanation:

Since **a** and **b** are pointing to 2 different objects, so the operator **is** returns **False**

a=2

b=2

c=a **is** b

print(c)

Output:

True

Explanation:

Since **a** and **b** are pointing to same objects, so the operator **is** returns **True**



Examples Of **is** Operator



```
a=2  
b=type(a) is int  
print(b)  
Output:  
True
```

Explanation:
type(a) is int evaluates
to **True** because 2 is indeed an
integer number.

```
a=2  
b=type(a) is float  
print(b)  
Output:  
False
```

Explanation:
type(a) is float evaluates
to **False** because 2 is not a **float**
number.



Examples Of **is not** Operator



```
a="Delhi"
```

```
b="Delhi"
```

```
c=a is not b
```

```
print(c)
```

Output:

False

Explanation:

Since **a** and **b** are pointing to the **same object**, so the operator **is not** returns **False**

```
a="Delhi"
```

```
b="delhi"
```

```
c=a is not b
```

```
print(c)
```

Output:

True

Explanation:

Since **a** and **b** are pointing to 2 **different objects**, so the operator **is not** returns **True**



Membership Operators



- Membership operators are used to test whether a **value** or **variable** is **found** in a **sequence** (**string**, **list**, **tuple**, **set** and **dictionary**).
- There are 2 Membership operators
 - **in**
 - **not in**



Behavior Of **in** and **not in**



- **in**: The '**in**' operator is used to check if a value exists in a **sequence** or not

- **not in** : The '**not in**' operator is the opposite of '**in**' operator. So, if a value **does not exists** in the **sequence** then it will return a **True** else it will return a **False**.



Examples Of in Operator



```
a="Welcome"  
b="om"  
print(b in a)
```

Output:
True

```
a="Welcome"  
b="mom"  
print(b in a)
```

Output:
False



Examples Of **not in** Operator



```
primes=[2,3,5,7,11]
```

```
x=4
```

```
print(x not in primes)
```

Output:

True

```
primes=[2,3,5,7,11]
```

```
x=5
```

```
print(x not in primes)
```

Output:

False



Precedence Of Operators



- There can be **more than one operator** in an **expression**.

- To evaluate these type of expressions there is a **rule** called **precedence** in all programming languages .

- It guides the **order** in which **operations** are **carried out**.

Precedence And Associativity



Operator	Name
()	Parentheses
**	Exponent
+x, -x	Unary plus, Unary minus
*, /, //, %	Multiplication, Division, Floor div, Mod
+, -	Addition, Subtraction
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR



Guess The Output



```
a=6/2+3**4  
print(a)
```

Output:

84.0

```
a=25/(2+3)**2  
print(a)
```

Output:

1.0

```
a=20-12//3**2  
print(a)
```

Output:

19



Associativity Of Operators



- When two operators have the same **precedence**, Python follows **associativity**

- **Associativity** is the **order** in which an **expression** is **evaluated** and almost all the operators have **left-to-right associativity**.



Associativity Of Operators



- For example, **multiplication** and **division** have the same **precedence**. Hence, if both of them are present in an expression, **left one is evaluated first**.

- Similarly , **exponentiation** (******) has **right to left** **associativity** .

- So if multiple ****** are present , they are solved from **right to left**.



Guess The Output



```
a=5*2//3  
print(a)
```

Output:

3

```
a=5*(2//3)  
print(a)
```

Output:

0



Guess The Output



```
a=2**3**2  
print(a)
```

Output:

512

Remember , ** has
Right to left
assoiativity

```
a=(2**3)**2  
print(a)
```

Output:

64



PYTHON
LECTURE 14



Today's Agenda



- **Input Function And Math Module In Python**
 - **Using the `input()` Function**
 - **Using the `math` module**
 - **Different ways of importing a module**
 - **Accepting multiple values in single line**

Accepting Input In Python



- To accept user input , **Python** provides us a **function** called **input ()**

- **Syntax:**

- **input([prompt])**
- The **input()** function takes a **single optional argument** , which is the **string** to be displayed on **console**.



Return Value Of `input()`



- The `input()` function reads a line from **keyboard**, converts the **line into a string** by removing the trailing newline, and **returns it**.

Example 1

(Using input() without message)



```
print("enter your name")
name=input()
print("Hello",name)
```

Output:

```
enter your name
Sachin
Hello Sachin
```

Example 2 (Using input() With Message)



```
name=input("enter your name")  
print("Hello",name)
```

□ Output:

```
enter your nameSachin  
Hello Sachin
```

Example 3 (Using input() With Message)



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

□ Output:

```
Enter your full name:Sachin Kapoor  
Hello Sachin Kapoor
```



Accepting Integer Input



- By default the **function input()** returns the inputted value as a **string**
- So , even if we input a numeric value , still **Python** considers it to be **string**



Accepting Integer Input



- To understand this behavior, consider the following code:

```
a=input("enter a number\n")  
b=a+1  
print(b)
```

- Output:

```
enter a number  
10  
Traceback (most recent call last):  
  File "inp_demo.py", line 2, in <module>  
    b=a+1  
TypeError: must be str, not int
```



Accepting Integer Input



- To **solve** this , we can use **Type Conversion Functions** in **Python** , for converting a given value from **string** to **other type**.

- For example , in the previous code , we can use the **function int()** to convert **string** value to **integer**



Accepting Integer Input



```
a=input("enter a number\n")
```

```
b=int(a)+1
```

```
print(b)
```

OR

```
a=int(input("enter a number\n")
```

```
b=a+1
```

```
print(b)
```

```
enter a number  
10  
11
```



Accepting Float And Bool



- For converting input values to **float** and **boolean** we can call **float()** and **bool()** functions

- Example:**

```
s=input("enter your percentage\n")
```

```
per=float(s)
```

```
print(per)
```

OR

```
s=input("Delete the file ?(yes-True,no-False)")
```

```
ans=bool(s)
```

```
print(ans)
```



Exercise



- **WAP to accept two numbers from the user and display their sum**

Code:

```
a=int(input("Enter first num:"))
b=int(input("Enter secnd num:"))
c=a+b
print("Nos are",a,"and",b)
print("Their sum is",c)
```

```
Enter first num:10
Enter secnd num:20
Nos are 10 and 20
Their sum is 30
```



Exercise



- Can you write the previous code in one line only ?

Code:

```
print("Their sum is",int(input("Enter first  
num:"))+int(input("Enter secnd num:")))
```

```
Enter first num:10  
Enter secnd num:20  
Their sum is 30
```



Exercise



- WAP to accept radius of a Circle from the user and calculate area and circumference.

Code:

```
radius=float(input("Enter radius:"))

area=3.14*radius**2
circum=2*3.14*radius

print("Area is",area)
print("Circumference is",circum)
```

```
Enter radius:5
Area is 78.5
Circumference is 31.400000000000002
```

Exploring More About math Module



- We have already discussed that **Python** has a **module** called **math** .

- This **module** helps us perform **mathematical calculations**

- It contains several **mathematical constants** and **functions**

Exploring More About math Module



- Following are some important functions :
 - `math.factorial(x)`
 - `math.floor(x)`
 - `math.ceil(x)`
 - `math.gcd(a, b)`
 - `math.pow(x,y)`
 - `math.sqrt(x)`

- Following are it's important mathematical constants:
 - `math.pi` : The mathematical constant $\pi = 3.141592\dots$
 - `math.e`: The mathematical constant $e = 2.718281\dots$,
 - `math.tau`: Tau is a circle constant equal to 2π

Modified Version Of Previous Code Using **math** Module



Code:

```
import math  
  
radius=float(input("Enter radius:"))  
  
area=math.pi*math.pow(radius,2)  
  
circum=math.tau*radius  
  
print("Area is",area)  
print("Circumference is",circum)
```

```
Enter radius:5  
Area is 78.53981633974483  
Circumference is 31.41592653589793
```

Second Way To Import A Module



- We can use **aliasing** for **module names**
- To do this , **Python** provides us **as keyword**
- **Syntax:**
 - **import modname as newname**
- This helps us to use **short names** for **modules** and make them **more easy** to use



Using **as** Keyword



Code

```
import platform as p  
print(p.system())
```

Windows

Second Way Of Writing Previous Code Using **math** Module



Code:

```
import math as m
radius=float(input("Enter radius:"))
area=m.pi*m.pow(radius,2)
circum=m.tau*radius
print("Area is",area)
print("Circumference is",circum)
```

```
Enter radius:5
Area is 78.53981633974483
Circumference is 31.41592653589793
```



Third Way To Import A Module



- We can also **import** specific members of a **module**
- To do this , **Python** provides us **from** keyword
- **Syntax:**
 - **from modname import name1[, name2[, ... nameN]]**
- In this way we will not have to **prefix the module** name before the **member name** while accessing it



Using **from** Keyword



Code

```
from sys import getsizeof
```

```
a=10
```

```
b="hello"
```

```
print(getsizeof(a))
```

```
print(getsizeof(b))
```

```
28
```

```
54
```

Third Way Of Writing Previous Code Using **math** Module



Code:

```
from math import pi,tau,pow
radius=float(input("Enter radius:"))
area=pi*pow(radius,2)
circum=tau*radius
print("Area is",area)
print("Circumference is",circum)
```

```
Enter radius:5
Area is 78.53981633974483
Circumference is 31.41592653589793
```



Fourth Way To Import A Module



- It is also **possible** to **import all names** from a **module** into the current file by using the **wildcard character ***

- **Syntax:**

- **from modname import ***
- This provides an **easy way** to **import** all the **members** from a **module** into the current file

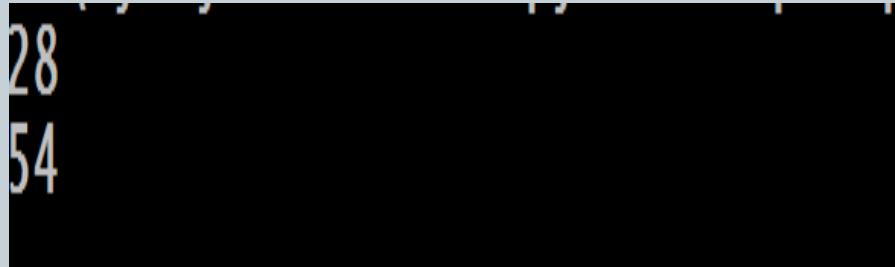


Using WildCard Character



Code

```
from sys import *
a=10
b="hello"
print(getsizeof(a))
print(getsizeof(b))
```

A black rectangular box representing a terminal window, containing the output of the provided Python code. The output shows two lines of text: '28' and '54', representing the memory size of integer and string variables respectively.

Fourth Way Of Writing Previous Code Using **math** Module



Code:

```
from math import *
radius=float(input("Enter radius:"))
area=pi*pow(radius,2)
circum=tau*radius
print("Area is",area)
print("Circumference is",circum)
```

```
Enter radius:5
Area is 78.53981633974483
Circumference is 31.41592653589793
```

How To List All Members Of A Module



- In **Python**, we can print **members** of a **module** in the **Python Shell window**
- This can be done in 2 ways:
 - By calling the **dir()** function passing it the **module name**
 - By calling the **help()** function passing it the **module name**



Using dir()

- The `dir()` function accepts the name of a **module** as **argument** and **returns a list** of all it's **members**.
- However the **module** must be **imported** before passing it to the `dir()` function

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fm
od', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'is
inf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan'
, 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```



Using help()



- The **help()** function **accepts the name of a module** as **argument** and **displays complete documentation** of all the **members** of the **module**
- Here also , **module** must be **imported** before using it.

```
>>> import math
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
```



Accepting Different Values



- WAP to accept roll number , grade and percentage as input from the user and display it back

Code

```
roll=int(input("Enter roll no:"))
name=input("Enter name:");
per=float(input("Enter per:"))
print("Roll no is",roll)
print("Name is",name)
print("Per is",per)
```

```
Enter roll no:10
Enter name:Sachin
Enter per:78.9
Roll no is 10
Name is Sachin
Per is 78.9
```



Exercise



- Write a program that asks the user to enter his/her name and age. Print out a message , displaying the user's name along with the year in which they will turn 100 years old.

```
What is your name?Sachin
How old are you?36
Hello Sachin
You will be 100 years old in the year 2083
```

- Hint: Use the module `datetime` to get the current year

Accepting Multiple Values In One Line



- In **Python**, the **input()** function can **read** and **return** a **complete line** of **input** as a **string**.

- However, we can split this input string into individual values by using the **function split()** available in the class **str**

- The **function split()**, breaks a **string** into **multiple strings** by using **space** as a **separator**

Accepting Multiple Values In One Line



- To understand , working of `split()` , consider the following example:

```
text="I Love Python"
```

```
word1,word2,word3=text.split()
```

```
print(word1)
```

```
print(word2)
```

```
print(word3)
```

Output:

I

Love

Python

Accepting Multiple Values In One Line



```
text=input("Type a 3 word message")
word1,word2,word3=text.split()
print("First word",word1)
print("Secnd word",word2)
print("Third word",word3)
```

Output:

```
Type a 3 word message:Good Morning Bhopal
First word Good
Secnd word Morning
Third word Bhopal
```



An Important Point!



- The **number of variables** on **left of assignment operator** and **number of values** generated by **split()** must be the **same**

```
Type a 3 word message:Good Morning Bhopal Indore
Traceback (most recent call last):
  File "inp_demo2.py", line 2, in <module>
    word1,word2,word3=text.split()
ValueError: too many values to unpack (expected 3)
```



Exercise



- Write a program that asks the user to input 2 integers and adds them . Accept both the numbers in a single line only

```
Enter 2 numbers:10 20
First number is 10
Second number is 20
Their sum is 30
```



Solution



Code:

```
s=input("Enter 2 numbers:")
a,b=s.split()
print("First number is",a);
print("Second number is",b)
c=int(a)+int(b)
print("Their sum is",c)
```

Accepting Multiple Values Separated With ,



- By default **split()** function considers , **space** as a **separator**
- However , we can **use** any **other symbol** also as a **separator** if we **pass** that **symbol** as **argument** to **split()** function
- **For example** , if we use **comma ,** as a **separator** then we can provide **comma separated input**



Example



Code:

```
s=input("Enter 2 numbers separated with comma:")  
a,b=s.split(",")  
print("First number is",a);  
print("Second number is",b)  
c=int(a)+int(b)  
print("Their sum is",c)
```

```
Enter 2 numbers separated with comma:2,4  
First number is 2  
Second number is 4  
Their sum is 6
```

Accepting Different Values In One Line



Code:

```
s=input("Enter roll no,name and per:")
roll,name,per=s.split()
print("Roll no is",roll)
print("Name is",name)
print("Per is",per)
```

```
Enter roll no,name and per:10 Sachin 78.9
Roll no is 10
Name is Sachin
Per is 78.9
```



PYTHON
LECTURE 15



Today's Agenda



- eval() Function , Command Line Arguments and Various print() Options
 - Using the eval() Function
 - Using Command Line Arguments
 - Using format specifiers in Python
 - Using the function format()



Using The Function `eval()`



- Python has a very interesting function called `eval()` .
- This function accepts any valid Python expression and **executes** it



Using The Function `eval()`



□ Syntax:

- `eval(expression)`

- The **argument** passed to `eval()` must **follow** below mentioned **rules:**
 - It must be given in the form of **string**
 - It should be a **valid Python code** or **expression**



Examples



- **Example:**

```
x = eval('2+3')  
print(x)
```

- **Output:**

5

- **Example:**

```
x = eval('2+3*6')  
print(x)
```

- **Output:**

20



Examples

- Example:
`eval('print(15)')`
- Output:
`15`
- Example:
`x=eval('print(15)')
print(x)`
- Output:
`15
None`



Examples

- **Example:**

```
x=eval('print()')  
print(x)
```

- **Output:**

None

- **Example:**

```
from math import sqrt  
x = eval('sqrt(4)')  
print(x)
```

- **Output:**

2.0

Using eval() For Type Conversion



- Another important use of `eval()` function is to perform **type conversion**.
- The `eval()` function **interprets the argument** inside character string and **converts it automatically** to its type.

Example:

```
x = eval('2.5')
print(x)
print(type(x))
```

Output:

```
2.5
<class 'float'>
```

Same Example

Without eval():

```
x = '2.5'
print(x)
print(type(x))
```

Output:

```
2.5
<class 'str'>
```



Using `eval()` With `input()`



- We can use `eval()` with `input()` function to perform automatic **type conversion** of values.

- In this way , we will not have to use **type conversion** functions like `int()` , `float()` or `bool()`



Example



Code:

```
age = eval(input("Enter your age "))  
age=age+10  
print("After 10 years , you will be ",age, "years old")
```

Output:

```
Enter your age 25  
After 10 years , you will be 35 years old
```



Guess The Output



- Example:

```
a=eval(input("Type something:"))
print(a)
print(type(a))
```

Suppose user types 25

- Output:

```
Type something:25
25
<class 'int'>
```



Guess The Output



- **Example:**

```
a=eval(input("Type something:"))
print(a)
print(type(a))
```

- Suppose user types 3.6

- **Output:**

```
Type something:3.6
3.6
<class 'float'>
```



Guess The Output



- **Example:**

```
a=eval(input("Type something:"))
print(a)
print(type(a))
```

- Suppose user types [10,20,30]

- **Output:**

```
Type something: [10, 20, 30]
[10, 20, 30]
<class 'list'>
```



Guess The Output



- **Example:**

```
a=eval(input("Type something:"))
print(a)
print(type(a))
```

- Suppose user types **Hello**

- **Output:**

```
Type something:Hello
Traceback (most recent call last):
  File "evaldemo1.py", line 1, in <module>
    a=eval(input("Type something:"))
  File "<string>", line 1, in <module>
NameError: name 'Hello' is not defined
```



Guess The Output



- Example:

```
a=eval(input("Type something:"))
print(a)
print(type(a))
```

- Suppose user types “Hello”

- Output:

```
Type something:"Hello"
Hello
<class 'str'>
```



Command Line Arguments



- Command Line Arguments are the **values**, we can **pass** while **executing** our **Python** code from **command prompt**

- **Syntax:**

- **python prog_name <values>**

These are called
**command line
arguments**

- **For example:**

- **python demo.py 10 20 30**

Their main benefit is that they are another mechanism to provide input to our program

Where Are Command Line Arguments Stored ?



- Command Line Arguments are stored by Python in a special predefined variable called argv

- Following are important features of argv:
 - This variable itself is stored in a module called sys

 - So to use it , we must import sys module in our program

 - The variable argv is actually a list

 - The name of the program is passed as the first argument which is stored at the 0th index in argv



Example

Code (suppose name of prog is cmdarg.py):

```
from sys import argv  
print(argv)  
print(type(argv))
```

Suppose we run it as **python cmdarg.py 10 20 30**

Output:

```
D:\My Python Codes>python cmdarg.py 10 20 30  
['cmdarg.py', '10', '20', '30']  
<class 'list'>
```



Accessing Individual Values



- **argv** is a **List** type object
- **Lists** are **index** based
- They always start from **0th** index
- So if we want to access individual elements of **argv** then we can use the **subscript operator** passing it the **index number**



Accessing Individual Values



Code:

```
from sys import argv  
print(argv[0])  
print(argv[1])  
print(argv[2])
```

Suppose we run it as **python cmdarg.py 10 20**

Output:

```
D:\My Python Codes>python cmdarg.py 10 20  
cmdarg.py  
10  
20
```



Guess The Output

Code:

```
from sys import argv  
print(argv[0])  
print(argv[1])  
print(argv[2])
```

If we try to access
argv beyond it's last
index then
Python will throw
IndexError exception

Execution: `python cmdarg.py`

Output:

`cmdarg.py`

`IndexError: list index out of range`

Obtaining Number Of Arguments Passed



- The built in function `len()` can be used to get the number of arguments passed from **command prompt**

Code:

```
from sys import argv  
n=len(argv)  
print("You have passed",n-1,"arguments")
```

Output:

```
D:\My Python Codes>python cmdarg.py  
You have passed 0 arguments
```

```
D:\My Python Codes>python cmdarg.py 10 20  
You have passed 2 arguments
```



Using Slicing Operator



- A **List** in **Python** is also a sequence type like a **string**
- So , it also supports **slicing** i.e. we can use the **slicing operator** `[:]`, to **retrieve the list values** from any **index**.
- For example , if we don't want the program name then we can use the slicing operator passing it the **index number 1** as **start index**



Example



Code (suppose name of prog is cmdarg.py):

```
from sys import argv  
print(argv[1:])
```

Execution: **python cmdarg.py 10 20 30**

Output:

```
D:\My Python Codes>python cmdarg.py 10 20 30  
['10', '20', '30']
```



Guess The Output



Code (suppose name of prog is cmdarg.py):

```
from sys import argv  
print(argv[1:])
```

Execution: python cmdarg.py

Output:

```
[ ]
```



Guess The Output



Code: addnos.py

```
from sys import argv  
print("First num is",argv[1])  
print("Sec num is",argv[2])  
print("Their sum is",argv[1]+argv[2])
```

Execution: python addnos.py 15 20

Output:

First num is 15

Sec num is 20

Their sum is 1520

By default , Python
treats all the command
line arguments as string
values



How To Solve This ?



- To solve the previous problem , we will have to **type convert** string values to **int**.

- This can be done by using **int()** or **eval()** function



Example



Code:

```
from sys import argv  
a=eval(argv[1])  
b=eval(argv[2])  
print("Nos are",a,"and",b)  
print("Their sum is",a+b)
```

```
D:\My Python Codes>python sumnos.py 10 20  
Nos are 10 and 20  
Their sum is 30
```



Guess The Output



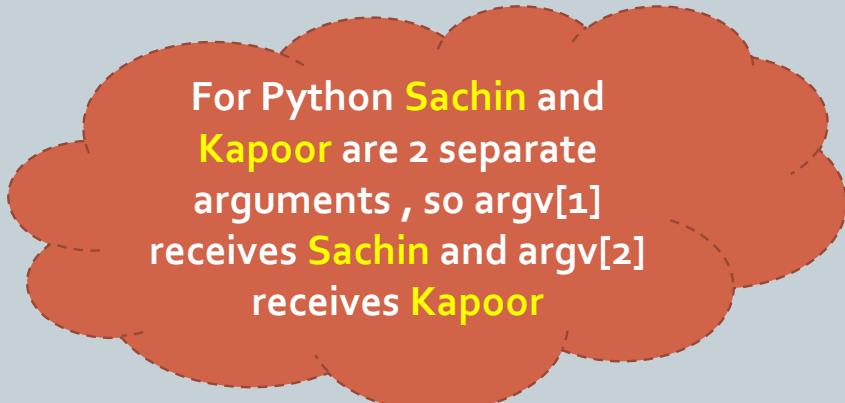
Code:

```
from sys import argv  
print("Hello",argv[1])
```

Execution: python cmdarg.py Sachin Kapoor

Output:

Hello Sachin



A large, orange, cloud-like thought bubble containing explanatory text. Three smaller orange dots are positioned to the left of the bubble, aligned vertically with the word "Hello" in the output text above it.

For Python Sachin and Kapoor are 2 separate arguments , so argv[1] receives Sachin and argv[2] receives Kapoor



Guess The Output



If we want to pass **Sachin Kapoor** as a **single argument** then we must enclose it in **double quotes**

Code:

```
from sys import argv  
print("Hello",argv[1])
```

Execution: **python cmdarg.py "Sachin Kapoor"**

Output:

Hello Sachin Kapoor



Guess The Output



Code:

```
from sys import argv  
print("Hello",argv[1])
```

Execution: **python cmdarg.py 'Sachin Kapoor'**

Output:

Hello 'Sachin'

On command prompt
only double quoted
strings are treated as
single value.

Using Format Specifiers With print()



- Just like **C** language **Python** also allows us to use **format specifiers** with **variables**.

- The **format specifiers** supported by **Python** are:
 - **%d:** Used for int values
 - **%i:** Used for int values
 - **%f:** Used for float values
 - **%s:** Used for string value
 - **%c:** Used for single char

Using Format Specifiers With print()



□ Syntax:

□ `print("format specifier" %(variable list))`

Example:

`a=10`

`print("value of a is %d " %(a))`

Output:

`value of a is 10`

If a single variable is
there then parenthesis
can be dropped

Using Format Specifiers With print()



Example:

a=10

msg="Welcome"

c=1.5

print("values are %d , %s,%f" %(a,msg,c))

Output:

Values are 10, Welcome, 1.500000

Number of format
specifiers must
exactly match with
the number of
values in the
parenthesis



Key Points About Format Specifier



- The **number of format specifiers** and **number of variables** must always **match**

- We should use the **specified format specifier** to display a **particular value**.

- **For example** we **cannot** use **%d** for **strings**

- However we **can use %s** with **non string** values also , like **boolean**



Examples



a=10

```
print("%s" %a)
```

Output:

10

a=10

```
print("%f" %a)
```

Output:

10.000000



Examples



a=10.6

```
print("%f" %a)
```

Output:

10.600000

a=10.6

```
print("%.2f" %a)
```

Output:

10.60

a=10.6

```
print("%d" %a)
```

Output:

10

a=10.6

```
print("%s" %a)
```

Output:

10.6



Examples



```
a=True  
print("%s" %a)
```

Output:

True

```
a=True  
print("%f" %a)
```

Output:
1.000000

```
a=True  
print("%d" %a)
```

Output:

1



Examples



```
a="Bhopal"  
print("%s" %a)
```

□ Output:

Bhopal

```
a="Bhopal"  
print("%f" %a)
```

Output:
TypeError

```
a="Bhopal"  
print("%d" %a)
```

□ Output:

TypeError: number required , not str



Examples



```
a="Bhopal"
```

```
print("%c" %a[0])
```

□ Output:

B

```
a="Bhopal"
```

```
print("%c" %a[0:2])
```

□ Output:

TypeError:

%c requires int or char

```
x=65
```

```
print("%c" %x)
```

Output:

A

```
x=65.0
```

```
print("%c" %x)
```

Output:

TypeError:

%c requires int or char



Using The Function `format()`



- Python 3 introduced a **new way** to do string formatting by providing a method called **format()** in **string** object
- This “**new style**” string formatting gets rid of the **%** operator and **makes the syntax** for string formatting **more regular**.



Using The Method `format()`



□ Syntax:

- `print("string with {}".format(values))`

□ Example

- `name="Sachin"`
- `age=36`
- `print("My name is {} and my age is {}".format(name,age))`

□ Output:

- `My name is Sachin and my age is 36`



Examples



```
name="Sachin"
```

```
age=36
```

```
print("My name is {1} and my age is {0}".format(age,name))
```

Output:

My name is Sachin and my age is 36



PYTHON
LECTURE 16



Today's Agenda



- **Decision Control Statements**
 - **The if Statement**
 - **Concept of Indentation**
 - **The if-else Statement**
 - **The if-elif-else Statement**
 - **What about ternary operator ?**



Decision Control Statements



- **Decision Control Statements** in **Python** are those statements which decide the execution flow of our program.

- In other words , they allow us to decide whether a **particular part of our program** should **run** or **not** based upon certain condition.

- The 4 **decision control statements** in **Python** are:
 - **if**
 - **if....else**
 - **if...elif...else**
 - **nested if**



The if Statement



- The **if** statement in **Python** is similar to other languages like in **Java**, **C**, **C++**, etc.

- It is used to **decide** whether a **certain statement** or **block of statements** will be executed or not .

- If a certain condition is **true** then a **block of statement** is **executed** otherwise **not**.



The if Statement



□ Syntax:

```
if (expression):
    statement1
    statement2
    .
    .
    .
    statement..n
```

□ Some Important Points:

- Python does not use {} to define the body of a code block , rather it uses indentation.
- A code block starts with indentation and ends with the first unindented line.
- The amount of indentation is up to the programmer, but he/she must be consistent throughout that block.
- The colon after if() condition is important and is a part of the syntax. However parenthesis with condition is optional



Exercise



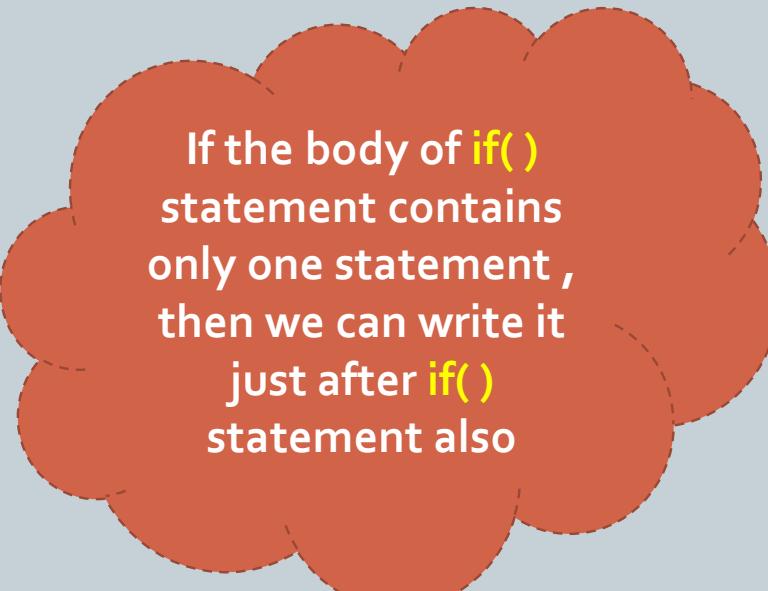
- **WAP to accept an integer from the user and check whether it is an even or odd number.**



Solution

Solution 1:

```
a=eval(input("Enter a number:"))
if(a%2==0):
    print("No is even")
if(a%2!=0):
    print("No is odd")
```



If the body of if() statement contains only one statement , then we can write it just after if() statement also

Solution 2:

```
if(a%2==0):print("No is even")
if(a%2!=0):print("No is odd")
```



What About Multiple Lines ?



- If there are **multiple lines** in the body of **if()** , then :
 - Either we can write them inside **if()** by properly indenting them
 - OR
 - If we write them just after **if()** , then we must use semicolon as a separator



What About Multiple Lines ?



Solution 1:

```
if(a%2==0):  
    print("No is even")  
    print("Hello")  
  
if(a%2!=0):  
    print("No is odd")  
    print("Hi")
```

Solution 2:

```
if(a%2==0): print("No is even");print("Hello")  
if(a%2!=0): print("No is odd");print("Hi")
```



The if –else Statement



- The **if..else** statement evaluates **test expression** and will execute body of **if** only when **test condition** is **True**.

- If the **condition** is **False**, body of **else** is executed.

- **Indentation** is used to **separate** the blocks.



The if-else Statement



□ Syntax:

```
if (expression):
    statement 1
    statement 2
else:
    statement 3
    statement 4
```

Indentation and **colon** are important for **else** also



Example

```
a=eval(input("Enter a number:"))
if(a%2==0):
    print("No is even")
else:
    print("No is odd")
```



Exercise



- WAP to accept a character from the user and check whether it is a capital letter or small letter. Assume user will input only alphabets



Solution

Solution 1:

```
ch=input("Enter a character:")
if "A" <= ch <= "Z":
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```

We also can use the logical and operator and make the conditions separate

Solution 2:

```
ch=input("Enter a character:")
if ch >= "A" and ch <= "Z":
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```



Guess The Output



Code:

```
ch=input("Enter a character:")
if 65<=ch<=90:
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```

Suppose the input given is A

Output:

TypeError: <= not supported between int and str



Why Did The Exception Occur ?



- Recall that, in **Python** we don't have **character data type** and even single letter data is a **string**.

- So the input "**A**" , is **not converted** to **UNICODE** automatically because it is still treated as a **string** value.

- Thus , the comparison failed between **string** and **integer**.



Solution



- The solution is to convert the "**A**" to it's corresponding **UNICODE** value .
- **Can you think , how can we do it ?**
- The answer is , using the function **ord()**.
- Recall that , this function accepts a **single letter string** and returns it's **UNICODE** value



Solution



```
ch=input("Enter a character:")
if 65<=ord(ch)<=90:
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```



The if –elif–else Statement



- The **elif** is short for **else if**. It allows us to check for multiple expressions.

- If the condition for **if** is **False**, it checks the condition of the next **elif** block and so on.

- If all the conditions are **False**, body of **else** is executed.



The **if –elif–else** Statement



□ Syntax:

```
if (expression):
    statement 1
    statement 2
elif (expression):
    statement 3
    statement 4
else:
    statement 5
    statement 6
```

Although it is not visible in the syntax , but we can have multiple **elif** blocks with a single **if** block



Exercise



- WAP to accept a character from the user and check whether it is a capital letter or small letter or a digit or some special symbol



Solution



```
ch=input("Enter a character:")
if "A" <=ch <="Z":
    print("You entered a capital letter")
elif "a" <=ch <="z":
    print("You entered a small letter")
elif "0" <=ch <="9":
    print("You entered a digit")
else:
    print("You entered some symbol")
```



The nested if Statement



- We can have a **if...elif...else** statement inside another **if...elif...else** statement.

- This is called **nesting** in computer programming.

- Any number of these statements can be nested inside one another.

- **Indentation is the only way to figure out the level of nesting**



The nested if Statement



□ Syntax:

```
if (expression):
    if (expression):
        statement 1
        statement 2
    else:
        statement 3
        statement 4
    statement 5
    statement 6
```



Exercise



- WAP to accept 3 integers from the user and without using any logical operator or cascading of relational operators , find out the greatest number amongst them



Solution



```
a,b,c=input("Enter 3 int").split()
a=int(a)
b=int(b)
c=int(c)
if a>b:
    if a>c:
        print("{} is greatest".format(a))
    else:
        print("{} is greatest".format(c))
else:
    if b>c:
        print("{} is greatest".format(b))
    else:
        print("{} is greatest".format(c))
```



Exercise



- **WAP to accept an year from the user and check whether it is a leap year or not.**

An year is a leap year if:

It is exactly divisible by 4 AND at the same time
not divisible by 100

OR

it is divisible by 400

For example:

2017 is not a leap year

2012 is a leap year

1900 is a not leap year

2000 is a leap year



Ternary Operator In Python



- Many **programming languages** have an **operator** called **ternary operator**, which is denoted by **? :**.
- It allows us to write complete **if – else** statement in just **one line**.
- For example , **C language** provides the following form of **ternary operator**:
 - **<condition> ? <expression1> : <expression2>**



Ternary Operator In Python



- But in **Python** we don't have a **ternary operator** officially .
- However **Python** provides us a **single line if – else** to work just like **ternary operator**
- Syntax:
 - **<expression1> if <condition> else <expression2>**
 - It first evaluates the **condition**; if it returns **True** then **expression1** will be evaluated to give the result, otherwise it will evaluate **expression2**.



Example



Example 1:

```
age=12
```

```
msg='Kid' if age<13 else 'Teenager'
```

```
print(msg)
```

Output:

```
Kid
```

Example 2:

```
age=16
```

```
msg='Kid' if age<13 else 'Teenager'
```

```
print(msg)
```

Output:

```
Teenager
```

These codes internally become:

```
if age<13:  
    msg='Kid'  
else:  
    msg='Teenager'
```



Example



Checking Even Odd Using Single Line if-else

```
a=eval(input("Enter a number:"))
msg= 'Even no' if a%2==0 else 'Odd No'
print(msg)
```

Output:

```
Enter a number:4
```

```
Even no
```

```
Enter a number:9
```

```
Odd No
```



Handling Multiple Conditions



- We can handle **multiple conditions** also **using single line if-else**

- Syntax:
 - <expression1> if <condition1> else <expression2> if <condition2> else <expression3>

- It first evaluates the **condition1**; if it returns **True** then **expression1** will be executed to give the result, otherwise it will evaluate the **condition2**; if it returns **True** then **expression2** will be executed otherwise it will execute **expression3**.



Example



Example 1:

age=16

msg='Kid' if age<13 else 'Teenager' if age<20 else 'Adult'

print(msg)

Output:

Teenager

Example 2:

age=21

msg='Kid' if age<13 else 'Teenager' if age<20 else 'Adult'

print(msg)

Output:

Adult

```
if age<13:  
    msg='Kid'  
else:  
    if age<20:  
        msg='Teenager'  
    else:  
        msg='Adult'
```



PYTHON
LECTURE 17



Today's Agenda



- **Iterative Statements**

- **Types of loop supported by Python**
- **The while loop**
- **The while-else loop**
- **The break , continue and pass Statement**

Iterative Statements



- There may be a situation when we need to **execute** a block of code **several number of times**.
 - For such situations , **Python** provides the concept of **loop**
 - A **loop** statement allows us to **execute a statement** or **group of statements multiple times**



Iterative Statements



- The **2 popular loops** provided by **Python** are:
 - **The while Loop**
 - **The for Loop**
- Recall that **Python** doesn't provide any **do..while** loop unlike **C,C++** and **Java**.



The **while** Loop



□ Syntax:

```
while condition:  
    <indented statement 1>  
    <indented statement 2>  
    ...  
    <indented statement n>  
    <non-indented statement 1>  
    <non-indented statement 2>
```

□ Some Important Points:

- First the condition is evaluated. If the condition is **true** then statements in the **while** block is **executed**.
- After executing statements in the **while** block the condition is checked again and if it is still **true**, then the statements inside the while block is **executed again**.
- The statements inside the **while** block will keep executing until the condition is **true**.
- Each execution of the loop body is known as **iteration**.
- When the condition becomes **false** loop terminates and program control comes **out of the while loop** to begin the **execution** of statement following it.

Examples

- Example 1:

```
i=1  
while i<=10:  
    print(i)  
    i=i+1  
print("done!")
```

- Output:

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

- Example 2:

```
i=1  
total=0  
while i<=10:  
    print(i)  
    total+=i  
    i=i+1  
print("sum is  
{0}".format(total))
```

- Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
sum is 55
```

Guess The Output



```
i=1  
while i<=10:  
    print(i)  
    i=i+1  
print("done!")
```

- ## • Output:

11111111

This is an infinite loop

```
i=1  
while i<=10:  
    print(i)  
    total+=i  
    i=i+1  
print("sum is  
{0}".format(total))
```

- ## • Output:

```
1
Traceback (most recent call last):
  File "loopdemo2.py", line 5, in <module>
    total=total+i
NameError: name 'total' is not defined
```



Another Form Of “**while**” Loop



- In **Python**, just like we have an **else** with **if**, similarly we also can have an **else** part with the **while** loop.

- The **statements** in the **else** part are **executed**, when the **condition is not fulfilled** anymore.



Another Form Of “while” Loop



□ Syntax:

while condition:

```
<indented statement 1>
<indented statement 2>
...
<indented statement n>
```

else:

```
<indented statement 1>
<indented statement 2>
```

□ Some Important Points:

- Many programmer's have a doubt that If the statements of the additional **else** part were placed right after the **while loop** without an **else**, they would have been executed anyway, wouldn't they.
- Then what is the use of **else**
- To understand this , we need to understand the **break** statement,



The “**break**” Statement



- Normally a **while** loop ends only when the **test condition** in the loop becomes **false**.

- However , with the help of a **break** statement a **while** loop can be left prematurely,

```
while test expression:  
    body of while  
    if condition:  
        break  
    body of while  
  
    ➔ statement(s)
```

Now comes the crucial point:
If a loop is left by **break,**
the else part is not executed.



Example

- Example 1:

```
i=1  
while i<=10:  
    if(i==5):  
        break  
    print(i)  
    i=i+1  
  
else:  
    print("bye")
```

Output:

```
1  
2  
3  
4
```

- Example 2:

```
i=1  
while i<=10:  
    print(i)  
    i=i+1  
  
else:  
    print("bye")
```

Output:

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



Exercise



- **WAP to accept a `string` from the `user` and `check whether it contains` any `vowel` or not.**

Sample Output:

```
Type a string:sachin  
sachin contains vowel
```

```
Type a string:rhythm  
rhythm doesn't contain any vowel
```



Exercise



- You have to develop a **number guessing game**. The program will generate a **random integer secretly**. Now it will ask the user to guess that number . If the user guessed it correctly then the program prints “**Congratulations! You guessed it right**” .

- If the **number guessed by the user** is **larger** than the **secret number** then program should print “**Number too large**” and , if the **number guessed by the user** is **smaller** than the **secret number** then program should print “**Number too small**” .

- This should continue until the **user guesses the number correctly** or **quits** . If the **user wants to quit** in between he will have to type **o** or **negative number**



Output



```
Guess the secret number:50
Your guess is too large. Try again!
Guess the secret number:30
Your guess is too large. Try again!
Guess the secret number:10
Your guess is too small. Try again!
Guess the secret number:20
Your guess is too small. Try again!
Guess the secret number:25
Your guess is too small. Try again!
Guess the secret number:27
Congratulations! You guessed it right!
```



Output



```
Guess the secret number:35
Your guess is too small. Try again!
Guess the secret number:70
Your guess is too small. Try again!
Guess the secret number:90
Your guess is too large. Try again!
Guess the secret number:0
So Sorry! That you are quitting!
```



How To Generate Random Number



- In **Python** , we have a module named **random** .

- This module contains a function called **randint()** , which accepts **2 arguments** and **returns** a **random number** between them (**both included**).

```
import random  
a=random.randint(1,20)  
print("Random number is",a)
```

Output:

Random number is 16



The “**continue**” Statement



- The **continue** statement in **Python** **returns the control** to the **beginning of the while loop**.
- It **rejects all the remaining statements** in the **current iteration** of the **loop** and **moves the control back** to the **top of the loop**.

```
while test expression:  
    # codes inside while loop  
    if condition:  
        continue  
        # codes inside while loop  
  
    # codes outside while loop
```



Example



```
i=0  
while i<10:  
    i=i+1  
    if(i%2!=0):  
        continue  
    print(i)
```

Output:

```
2  
4  
6  
8  
10
```



Exercise



- Write a program to accept a **string** from the user and **display it vertically** but **don't display the vowels** in it.

- Sample Output:

```
Type a string:Sachin
S
c
h
n
```



Exercise



- Write a program to continuously accept integers from the user until the user types 0 and as soon as 0 is entered display sum of all the nos entered before 0

- Sample Output:

```
Enter an integer(press 0 to stop):5
Enter an integer(press 0 to stop):2
Enter an integer(press 0 to stop):11
Enter an integer(press 0 to stop):6
Enter an integer(press 0 to stop):0
Sum is 24
```



Exercise



- **Modify** the **previous code** so that if the **user** inputs **negative integer**, your program should **ignore it** .

- **Sample Output:**

```
Enter an integer(press 0 to stop):5
Enter an integer(press 0 to stop):2
Enter an integer(press 0 to stop):-6
Enter an integer(press 0 to stop):11
Enter an integer(press 0 to stop):0
Sum is 18
```



The “**pass**” Statement



- In **Python**, the **pass** statement is a no operation statement.
- That is , **nothing happens** when **pass** statement is **executed**.
- Example:

```
if (num == 15):
    #write_your_code and remove pass
    pass
elif(num==18):
    break
```

This will prevent the code from syntax error.



Example



```
i=0  
while i<10:  
    i=i+1  
    if(i%2!=0):  
        pass  
    else:  
        print(i)
```

Output:

```
2  
4  
6  
8  
10
```



PYTHON
LECTURE 18



Today's Agenda



- **The for Loop**
 - **The for Loop In Python**
 - **Differences with other languages**
 - **The range() Function**
 - **Using for with range()**



The **for** Loop



- Like the **while** loop the **for** loop also is a **programming language statement**, i.e. an **iteration statement**, which **allows a code block** to be **executed multiple number of times**.
- There are **hardly programming languages** without **for** loops.
- However the **for** loop **exists in many different flavours**, i.e. both the **syntax** and the **behaviour differs from language to language**



The **for** Loop



- Different Flavors Of “for” Loop:
- Count-controlled for loop (Three-expression for loop):

□ This is **by far** the **most common** type. This statement is the one used by **C**, **C++** and **Java**. Generally it has the form:

for (i=1; i <= 10; i++)

This kind of for loop is not implemented in Python!

- Numeric Ranges
 - This kind of **for loop** is a **simplification of the previous** kind. Starting with a **start value** and **counting up** to an **end value**, like
 - **for i = 1 to 100**
Python doesn't use this either.



The **for** Loop



□ Iterator-based for loop

- Finally, we come to the one used by **Python**. This kind of a **for** loop **iterates** over a **collection** of **items**.
- In **each iteration** step a **loop variable** is set to a **value** in a **sequence** or other **data collection**.
- This kind of **for loop** is known in most **Unix** and **Linux** shells and it is the one which is implemented in **Python**.



Syntax Of **for** Loop In Python



□ Syntax:

```
for some_var in some_collection:  
    # loop body  
    <indented statement 1>  
    <indented statement 2>  
    ...  
    <indented statement n>  
<non-indented statement 1>  
<non-indented statement 2>
```

□ Some Important Points:

- The **for** loop in **Python** can iterate over **string , list , tuple , set,frozenset , bytes,bytearray** and **dictionary**
- The **first item** in the collection is assigned to the **loop variable**.
- Then the block is **executed**.
- Then again the **next item of collection** is assigned to the **loop variable**, and the statement(s) block is **executed**
- This goes on until the entire **collection** is **exhausted**.



Examples

- **Example 1:**

```
word="Sachin"  
for ch in word:  
    print(ch)
```

- **Output:**

```
S  
a  
c  
h  
i  
n
```

- **Example 2:**

```
fruits=["Apple","Bana  
na","Guava","Ora  
nge"]  
for fruit in fruits:  
    print(fruit)
```

- **Output:**

```
Apple  
Banana  
Guava  
Orange
```



Exercise



- Write a program using **for** loop to accept a **string** from the **user** and **display it vertically** but **don't display the vowels** in it.

- Sample Output:

```
Type a string:Sachin
S
c
h
n
```

QUIZ- Test Your Skills



1. What is the output ?

```
word="sachin"  
if(ch in ["a","e","i","o","u"]):  
    continue  
print(ch,end=" ")
```

- A. s c h n
- B. Error
- C. sachin
- D. Exception

Correct Answer: B



QUIZ- Test Your Skills



2. What is the output?

i=0

```
while i<4:  
    i=i+1  
    if(i%2!=0):  
        pass  
        print("hi",end=" ")  
    else:  
        print(i,end=" ")
```

- A. hi 2 hi 4
- B. Syntax Error
- C. 2 4
- D. Infinite loop

Correct Answer: A

QUIZ- Test Your Skills



3. What is the output?

```
i=0
while i<4:
    i=i+1
    if(i%2!=0):
        continue
        print("hi",end=" ")
    else:
        print(i,end=" ")
```

- A. hi 2 hi 4
- B. Syntax Error
- C. 2 4
- D. Infinite loop

Correct Answer: C



QUIZ- Test Your Skills



4. What is the output?

i=0

```
while i<4:  
    i=i+1  
    if(i%2!=0):  
        break  
        print("hi",end=" ")  
    else:  
        print(i,end=" ")
```

- A. hi 2 hi 4
- B. Syntax Error
- C. 2 4
- D. No output

Correct Answer: D

QUIZ- Test Your Skills



5. What is the output ?

```
x = 123  
for i in x:  
    print(i)
```

- A. 123
- B. 1
- 2
- 3
- C. TypeError
- D. Infinite loop

Correct Answer: C

QUIZ- Test Your Skills



6. What is the output ?

i = 1

while True:

```
    if i%3 == 0:  
        break  
    print(i,end=" ")  
    i += 1
```

- A. Syntax Error
- B. 1 2
- C. 1 2 3
- D. Blank Screen(No Output)

Correct Answer: A



QUIZ- Test Your Skills

7. What is the output ?

i = 1

while True:

 if i%2 == 0:

 break

 print(i,end="")

 i += 2

A. 1

B. 1 2

C. Infinite loop

D. Syntax Error

Correct Answer: C

QUIZ- Test Your Skills



8. What is the output ?

```
x = "abcdef"
```

```
i = "i"
```

```
while i in x:
```

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

- A. abcdef
- B. ii iii i
- C. Error
- D. No output

Correct Answer: D

QUIZ- Test Your Skills



9. What is the output ?

x = "abcdef"

i = "a"

while i in x:

 print(i, end=" ")

- A. a b c d e f
- B. Infinite loop
- C. Error
- D. No output

Correct Answer: B

QUIZ- Test Your Skills



10. What is the output ?

```
x = "abcdef"
```

```
i = "a"
```

```
while i in x:
```

```
    x = x[1:]
```

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

- A. a a a a a
- B. a
- C. Error
- D. No output

Correct Answer: B

QUIZ- Test Your Skills



11. What is the output ?

```
x = 'abcd'  
for i in x:  
    print(i,end=" ")  
    x.upper()
```

- A. a B C D
- B. A B C D
- C. a b c d
- D. Syntax Error

Correct Answer: C

QUIZ- Test Your Skills



12. What is the output ?

```
x = 'abcd'  
for i in x:  
    print(i.upper())
```

- A. a B C D
- B. A B C D
- C. a b c d
- D. Syntax Error

Correct Answer: B

QUIZ- Test Your Skills



13. What is the output ?

```
text = "my name is sachin"
```

```
for i in text:
```

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

- A. my,name,is,sachin,
- B. m,y,,n,a,m,e,,i,s,,s,a,c,h,i,n,
- C. Syntax Error
- D. No output

Correct Answer: B

QUIZ- Test Your Skills



14. What is the output ?

```
text = "my name is sachin"  
for i in text.split():  
    print (i, end=", ")
```

- A. my,name,is,sachin,
- B. m,y,n,a,m,e,i,s,s,a,c,h,i,n
- C. Syntax Error
- D. No output

Correct Answer: A

QUIZ- Test Your Skills



15. What is the output ?

```
text = "my name is sachin"  
for i not in text:  
    print (i, end=", ")
```

- A. my,name,is,sachin,
- B. m,y,n,a,m,e,i,s,s,a,c,h,i,n
- C. Syntax Error
- D. No output

Correct Answer: C

QUIZ- Test Your Skills



16. What is the output?

True = False

while True:

 print(True)

 break

- A. True
- B. False
- C. No output(Blank Screen)
- D. None of the above

Correct Answer: D

QUIZ- Test Your Skills



17. What is the output?

i = 2

while True:

 if i%3 == 0:

 break

 print(i,end=" ")

 i += 2

- A. Infinite loop
- B. 2 4
- C. 2 3
- D. None of the above

Correct Answer: B



The **range** Function



- The **range()** function is a **built-in function** in **Python**, and it returns a **range** object.

- This **function** is **very useful** to generate a **sequence of numbers** in the form of a **List**.

- The **range()** function takes **1** to **3** arguments

The **range** Function With One Parameter



□ Syntax:

□ **range(n)**

□ For an argument **n**, the function returns a **range** object containing integer values from **0** to **n-1**.

Example:

```
a=range(10)  
print(a)
```

As we can see that when we display the variable **a** , we get to see the description of the **range** object and not the values.

Output:

```
range(0, 10)
```

To see the values , we must convert **range** object to **list**

The **range** Function With One Parameter



Example:

```
a=range(10)  
b=list(a)  
print(b)
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The function **list()** accepts a **range** object and **converts it** into a **list** of values .

These values are the numbers from **0** to **n-1** where **n** is the argument passed to the function **range()**

What If We Pass Negative Number ?



Guess:

```
a=range(-10)  
b=list(a)  
print(b)
```

Output:



The output is an **empty list** denoted by **[]** and it tells us that the function **range()** is coded in such a way that it always moves towards **right side** of the **start value** which here is **0**.

But since **-10** doesn't come **towards right** of **0**, so the output is an **empty list**

The **range** Function With Two Parameter



□ Syntax:

- **range(m,n)**

- For an argument **m,n** , the function returns a **range** object containing **integer values** from **m** to **n-1**.

Example:

```
a=range(1,10)  
print(a)
```

Output:

```
range(1, 10)
```

Here again when we display the variable **a** , we get to see the description of the **range** object and not the values. So we must use the function **list()** to get the values

The range Function With Two Parameter



Example:

```
a=range(1,10)
```

```
b=list(a)
```

```
print(b)
```

Output:

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

The output is **list** of numbers from **1** to **9**
because **10** falls **towards right** of **1**

What If We Pass First Number Greater?



Guess:

```
a=range(10,1)  
b=list(a)  
print(b)
```

Output:

```
[]
```

The output is an **empty list** because **as mentioned earlier** it **traverses towards right** of start value and **1** doesn't come to the right of **10**



Passing Negative Values



- We can pass **negative start** or/and **negative stop value** to **range()** when we call it with **2 arguments** .

Example:

```
a=range(-10,3)  
b=list(a)  
print(b)
```

Since **3** falls on **right of -10** ,
so we are **getting range of numbers from
-10 to 3**

Output:

```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2]
```



Guess The Output



```
a=range(-10,-3)  
b=list(a)  
print(b)
```

□ Output:

```
[-10, -9, -8, -7, -6, -5, -4]
```

```
a=range(-3,-10)  
b=list(a)  
print(b)
```

□ Output:

```
[]
```

```
a=range(-3,-3)  
b=list(a)  
print(b)
```

Output:

```
[]
```

The **range** Function With Three Parameter



□ Syntax:

- **range(m,n,s)**
- Finally, the **range()** function can also take the **third parameter** .
This is for the **step value**.

Example:

```
a=range(1,10,2)
```

```
b=list(a)
```

```
print(b)
```

Since **step value** is **2** , so we got nos from **1** to **9** with a **difference** of **2**

Output:

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



Guess The Output

```
a=range(7,1,-2)  
b=list(a)  
print(b)
```

Output:

[7, 5, 3]

```
a=range(5,10,20)  
b=list(a)  
print(b)
```

Output:

[5]

Pay close attention ,
that we are having
start value greater than
end value , but since
step value is negative ,
so it is allowed

Here, note that the
first integer, **5**, is
always returned, even
though the interval **20**
sends it beyond **10**



Guess The Output

```
a=range(2,14,1.5)  
b=list(a)  
print(b)
```

Output:

`TypeError: 'float' object cannot be interpreted as an integer`

Note that all three arguments must be integers only.

```
a=range(5,10,0)  
b=list(a)  
print(b)
```

Output:

`ValueError: range() arg 3 must not be zero`

It raised a **ValueError** because the interval cannot be **zero** if we need to go from one number to another.



Guess The Output

```
a=range(2,12)  
b=list(a)  
print(b)
```

Output:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
a=range(12,2)  
b=list(a)  
print(b)
```

Output:

```
[]
```

The default value of step is 1 , so the output is from 2 to 11

As usual , since the start value is greater than end value so we get an empty list



Using `range()` With `for` Loop



- We can use `range()` and `for` together for **iterating** through a **list** of **numeric values**
- **Syntax:**
 - `for <var_name> in range(end):`
 - indented statement 1
 - indented statement 2
 -
 -
 - indented statement n



Example



Code:

```
for i in range(11):  
    print(i)
```

Output:

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Using 2 Parameter range() With for Loop



- We can use 2 argument `range()` with `for` also for iterating through a list of **numeric values** between a **given range**

Syntax:

- `for <var_name> in range(start,end)`

- indented statement 1

- indented statement 2

- .

- .

- indented statement n



Example



Code:

```
for i in range(1,11):  
    print(i)
```

Output:

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



Exercise



- Write a program to accept an integer from the user and display the sum of all the numbers from 1 to that number.

- Sample Output:

```
Enter an int:5
sum of nos from 1 to 5 is 15
```



Solution



```
num=int(input("Enter an int:"))

total=0

for i in range(1,num+1):
    total=total+i

print("sum of nos from 1 to {} is {}".format(num,total))
```



Exercise



- Write a program to accept an integer from the user and calculate it's factorial

- Sample Output:

```
Enter an int:6  
Factorial of 6 is 720
```

Using 3 Parameter **range()** With **for** Loop



□ Syntax:

- `for <var_name> in range(start,end,step)`
 - indented statement 1
 - indented statement 2
 -
 -
 - indented statement n



Example



Code:

```
for i in range(1,11,2):  
    print(i)
```

Output:

```
1  
3  
5  
7  
9
```



Example



Code:

```
for i in range(100,0,-10):  
    print(i)
```

Output:

```
100  
90  
80  
70  
60  
50  
40  
30  
20  
10
```



Using `for` With `else`



- Just like `while`, the `for` loop can also have an `else` part, which **executes** if no `break` statements executes in the `for` loop

- **Syntax:**

```
for <var_name> in some_seq:  
    indented statement 1  
    if test_cond:  
        break  
else:  
    indented statement 3  
    indented statement 4
```



Example



Code:

```
for i in range(10):  
    print(i)  
else:  
    print("Loop complete")
```

Output:

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
Loop complete
```



Example



Code:

```
for i in range(1,10):  
    print(i)  
    if i%5==0:  
        break  
    else:  
        print("Loop complete")
```

Output:

```
1  
2  
3  
4  
5
```



Using Nested Loop



- Loops can be **nested** in **Python**, as they can with **other programming languages**.
- A **nested loop** is a **loop** that **occurs within another loop**, and are constructed like so:

Syntax:

```
for <var_name> in some_seq:  
    for <var_name> in some_seq:  
        indented statement 1  
        indented statement 2
```



Example



Code:

```
numbers = [1, 2, 3]
alpha = ['a', 'b', 'c']
for n in numbers:
    print(n)
    for ch in alpha:
        print(ch)
```

Output:

```
1
a
b
c
2
a
b
c
3
a
b
c
```

Exercise



- Write a program to print the following pattern

Sample Output:



* * *

* * *

* * *

* * *



Solution



Code:

```
for i in range(1,5):
    for j in range(1,4):
        print("*",end="")
    print()
```

Output:

A black rectangular box representing a terminal window. Inside, there are four lines of text, each containing three asterisks (*). The first three lines have an additional asterisk at the beginning of the line.

```
* * *
* * *
* * *
* * *
```



Solution

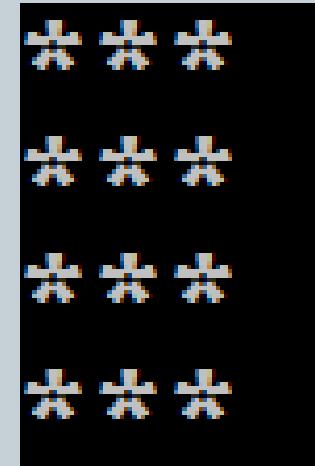


Can you **write the same code** using **only**
single loop ?

Output:

Code:

```
for i in range(1,5):  
    print("*"*3)
```

A 4x3 grid of asterisks on a black background. Each row contains three asterisks (*), and there are four such rows, representing the output of the provided Python code.

Exercise



- Write a program to print the following pattern

Sample Output:





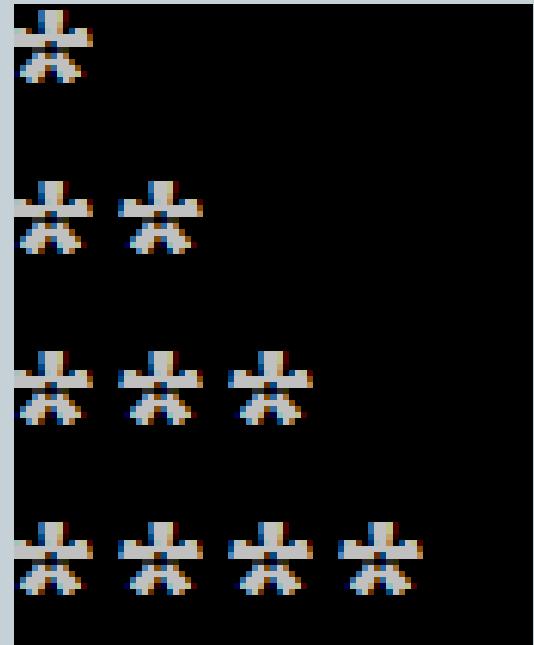
Solution



Code:

```
for i in range(1,5):
    for j in range(1,i+1):
        print("*",end="")
    print()
```

Output:



Exercise



- Write a program to print the following pattern

Sample Output:

```
*****
 ****
  **
   *

```



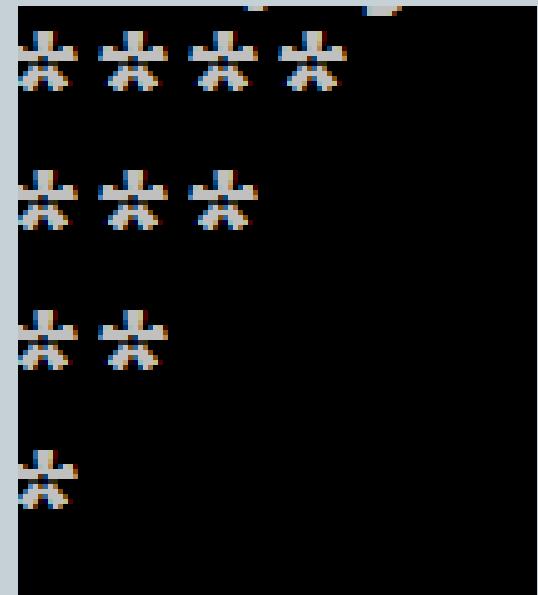
Solution



Code:

```
for i in range(4,0,-1):
    for j in range(1,i+1):
        print("*",end="")
    print()
```

Output:





Exercise



- Write a program to accept an integer from the user and display all the numbers from 1 to that number. Repeat the process until the user enters 0.
- Sample Output:

```
Enter a number: 3
```

```
1  
2  
3
```

```
Enter a number: 9
```

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

```
Enter a number: 0
```



Solution



Code:

```
x = int(input('Enter a number: '))  
while x != 0:  
    for y in range (1, x+1):  
        print (y)  
    x = int(input('Enter a number: '))
```

Output:

```
Enter a number: 3  
1  
2  
3  
Enter a number: 9  
1  
2  
3  
4  
5  
6  
7  
8  
9  
Enter a number: 0
```



PYTHON

LECTURE 19



Today's Agenda



- **User Defined Functions**
 - What Is A Function ?
 - Function V/s Method
 - Steps Required For Developing User Defined Function
 - Calling A Function
 - Returning Values From Function



What Is A Function ?



- A **function** in Python is a **collection of statements** having a **particular name** followed by **parenthesis** .

- To **run** a function , we have to **call** it and when we call a function **all the statements** inside the **function** are **executed**.

- So we don't have to write the code again and again

- This is called **code re-usability**.



Function V/s Method



- **Functions** are block of codes defined **individually** .
- But if a function is **defined inside a class** , it becomes a **method**
- So , **methods** and **functions** are same except their placement in the program.
- Also we can call a **function** directly using it's **name** but when we call a **method** we have to use either **object name** or **class name** before it



Function V/s Method



For example:

- `print("hello")`

- Here **print()** is a function as we are calling it directly
 - `message="Good Morning"`
 - `print(message.lower())`

- Here **lower()** is a method which belongs to the class **str** and so it is called using the object **message**



Steps Required For Function



- To create and use a function we have to take **2 steps**:
- **Function Definition**: Creating or writing the body of a function is called defining it. It contains the set of statements we want to run , when the function execute.
- **Function Call**: A function never runs automatically . So to execute it's statements we must call it



Syntax Of Function Definition

```
def function_name(param 1,param 2,...):  
    statement(s)
```

- Keyword **def** marks the start of **function header**.
- It is followed by a **function name** to uniquely identify it.
- **Parameters** (arguments) through which we pass values to a function. They are **optional**.
- A **colon** (:) to mark the end of **function header**.
- One or more valid **python statements** that make up the function body . All the statements must have same indentation level

Example Of Function Definition



```
def add(a,b):  
    print("Values are",a,"and",b)  
    c=a+b  
    print("Their sum is",c)
```



How To Call A Function ?



- Once we have **defined** a function, we can **call** it from another **function, program** or even the **Python prompt**.

- To call a function we simply type the function name with appropriate parameters.

- **Syntax:**
 - **function_name(arguments)**



Complete Example

```
def add(a,b):  
    print("Values are",a,"and",b)  
    c=a+b  
    print("Their sum is",c)  
add(5,10)  
add(2.5,5.4)
```

Output:

```
Values are 5 and 10  
Their sum is 15  
Values are 2.5 and 5.4  
Their sum is 7.9
```

Returning Values From Function



- To return a value or values from a function we have to write the keyword **return** at the end of the function body along with the value(s) to be returned

□ Syntax:

- **return <expression>**



Complete Example

```
def add(a,b):  
    c=a+b  
    return c  
  
x=add(5,10)  
print("Sum of 5 and 10 is",x)  
  
y=add(2.5,5.4)  
print("Sum of 2.5 and 5.4 is",y)
```

Output:

```
Sum of 5 and 10 is 15  
Sum of 2.5 and 5.4 is 7.9
```



Exercise



- Write a function called **absolute()** to accept an integer as argument and return it's **absolute value**. Finally **call** it to get the absolute value of **-7** and **9**

- Sample Output:

```
absolute of -7 is 7  
absolute of 9 is 9
```



Solution



```
def absolute(n):  
    if n>0:  
        return n  
    else:  
        return -n
```

```
x=absolute(-7)  
print("absolute of -7 is",x)  
y=absolute(9)  
print("absolute of 9 is",y)
```



Exercise



- Write a function called **factorial()** which accepts a number as argument and returns it's factorial. Finally call the function to calculate and return the factorial of the number given by the user.

```
□ Enter an int:4  
Factorial of 4 is 24
```



Solution



```
def factorial(n):
    f=1
    while n>1:
        f=f*n
        n=n-1
    return f
```

```
x=int(input("Enter an int:"))
y=factorial(x)
print("Factorial of",x,"is",y)
```



Guess The Output



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

```
greet("sachin")  
greet()
```

Output:

```
Hello sachin  
Traceback (most recent call last):  
  File "func5.py", line 5, in <module>  
    greet()  
TypeError: greet() missing 1 required positional argument: 'name'
```



Guess The Output



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

```
greet("sachin", "amit")
```

Output:

```
TypeError: greet() takes 1 positional argument but 2 were given
```



Guess The Output



```
def greet(name):  
    print("Hello",name)  
    return  
    print("bye")
```

```
greet("sachin")
```

Output:

```
Hello sachin
```



Guess The Output



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

```
x=greet("sachin")  
print("value in x is",x)
```

Output:

```
Hello sachin  
value in x is None
```

Returning Multiple Values From Function



- In languages like **C** or **Java**, a function can return only one value. However in **Python**, a function can return **multiple values** using the following syntax:
- Syntax:
 - `return <value 1, value 2, value 3, ... >`
- For example:
 - `return a,b,c`
- When we do this, **Python** returns these values as a **tuple**, which just like a **list** is a collection of multiple values.



Receiving Multiple Values



- To receive multiple values returned from a function , we have **2 options:**
- **Syntax 1:**
 - `var 1,var 2,var 3=<function_name>()`
- **Syntax 2:**
 - `var=<function_name>()`
- In the **first case** we are receiving the values in **individual variables** . Their **data types** will be set according to the **types of values** being **returned**
- In the **second case** we are receiving it in a **single variable** and **Python** will automatically make the data type of this variable as **tuple**



Complete Example

```
def calculate(a,b):  
    c=a+b  
    d=a-b  
    return c,d
```

```
x,y=calculate(5,3)
```

```
print("Sum is",x,"and difference is",y)
```

```
z=calculate(15,23)
```

```
print("Sum is",z[0],"and difference is",z[1])
```

Output:

```
Sum is 8 and difference is 2  
Sum is 38 and difference is -8
```

Here Python will automatically set x and y to be of int type and z to be of tuple type



PYTHON

LECTURE 20



Today's Agenda



- **User Defined Functions-II**
 - Arguments V/s Parameters
 - Types Of Arguments



Parameters V/s Arguments?



- A lot of people—mix up **parameters** and **arguments**, although they are slightly different.

- A **parameter** is a variable in a **method definition**.

- When a method is called, the **arguments** are the data we pass into the method's **parameters**.



Parameters V/s Arguments?



```
def multiply(x,y):  
    print(x*y)
```

parameters

```
multiply(2,8)
```

arguments



Types Of Arguments



- In **Python** , a function can have **4** types of arguments:
 - **Positional Argument**
 - **Keyword Argument**
 - **Default Argument**
 - **Variable Length Argument**



Positional Arguments



- These are the arguments passed to a function in correct **positional order**.
- Here the number of **arguments** in the call must exactly match with number of **parameters** in the function definition



Positional Arguments

```
def attach(s1,s2):  
    s3=s1+s2  
    print("Joined String is:",s3)  
  
attach("Good","Evening")
```

These are called
**POSITIONAL
ARGUMENTS**

Output:

```
Joined String is: GoodEvening
```



Positional Arguments



- If the **number of arguments** in call do not match with the **number of parameters** in function then we get **TypeError**:

```
def attach(s1,s2):  
    s3=s1+s2  
    print("Joined String is:",s3)
```

attach("Good")

Output:

```
TypeError: attach() missing 1 required positional argument: 's2'
```



Guess The Output



```
def grocery(name,price):  
    print("Item is",name,"It's price is",price)
```

```
grocery("Bread",20)  
grocery(150,"Butter")
```

Output:

```
Item is Bread It's price is 20  
Item is 150 It's price is Butter
```

The Problem With Positional Arguments



- The problem with **positional arguments** is that they always **bind** to the **position** of parameters.
- So **1st argument** will be copied to **1st parameter** , **2nd argument** will be copied to **2nd parameter** and so on.
- Due to this in the previous example the value **150** was copied to **name** and “**Butter**” was copied to **price**
- To solve this problem , **Python** provides us the concept of **keyword arguments**



Keyword Arguments



- **Keyword arguments** are arguments that identify parameters with their names

- With **keyword arguments** in **Python**, we can change the order of passing the arguments without any consequences

- Syntax:
function_name(paramname1=value,paramname2=value)



Complete Example

```
def grocery(name,price):  
    print("Item is",name,"It's price is",price)
```

```
grocery(name="Bread",price=20)  
grocery(price=150,name="Butter")
```

Output:

```
Item is Bread It's price is 20  
Item is Butter It's price is 150
```



Point To Remember!



- A **positional argument** can never follow a **keyword argument** i.e. the **keyword argument** should always appear after **positional argument**

- **For example:**

- `def display(num1,num2):`
 - `# some code`

Now if we call the above function as:

`display(10,num2=15)`

Then it will be **correct**. But if we call it as:

`display(num1=10,15)`

Then it will be a **Syntax Error**



Default Arguments



- For some functions, we may want to make some parameters ***optional*** and use **default values** in case the user does not want to provide values for them.
- This is done with the help of **default argument** values.
- We can specify **default argument** values for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the **default value**.
- Syntax:
def function_name(paramname1=value,paramname2=value):
 # function body



Complete Example

```
def greet(name,msg="Good Morning"):  
    print("Hello",name,msg)
```

```
greet("Sachin")  
greet("Amit","How are you?")
```

Output:

```
Hello Sachin Good Morning  
Hello Amit How are you?
```



Point To Remember!



- A function can have **any number of default arguments** but once we have a **default argument**, all the arguments to **it's right must also have default values**.

- This means to say, **non-default arguments** cannot follow **default arguments**.



Point To Remember!



- **For example:** if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

- Then we would have got the following **SyntaxError**

```
def greet(msg="Good Morning",name):  
    ^
```

```
SyntaxError: non-default argument follows default argument
```



Point To Remember!



- If a function has **default arguments** , set then while calling it if we are **skipping** an argument then **we must skip all the arguments after it also.**

- **For example:**

```
def show(a=10,b=20,c=30):  
    print(a,b,c)
```

- Now , if we call the above function as :

show(5)

- It will work and output will be **5 20 30**

- If we call it as :

show(5,7)

- Still it will work and output will be **5 7 30**

- But if we call it as

show(5, ,7)

- Then it will be an error

The solution to this problem is to use **default argument** as **keyword argument** :

show(5,c=7)

This will give the output as

5 20 7



Exercise



- Write a function called **cal_area()** using **default argument** concept which accepts **radius** and **pi** as arguments and calculates and displays area of the Circle. The value of **pi** should be used as **default argument** and value of **radius** should be **accepted from the user**



Solution



```
def cal_area(radius,pi=3.14):
    area=pi*radius**2
    print("Area of circle with radius",radius,"is",area)
```

```
rad=int(input("Enter radius:"))
cal_area(rad)
```

```
Enter radius:4
Area of circle with radius 4 is 50.24
```



Guess The Output ?



```
def addnos(a,b):  
    c=a+b  
    return c
```

```
def addnos(a,b,c):  
    d=a+b+c  
    return d  
  
print(addnos(10,20))  
print(addnos(10,20,30))
```

Output:

```
print(addnos(10,20))  
TypeError: addnos() missing 1 required positional argument: 'c'
```



Why Did The Error Occur ?



- The error occurred because **Python** does not support **Function or Method Overloading**
- **Moreover Python understands the latest definition of a function addnos() which takes 3 arguments**
- Now since we passed **2 arguments** only , the call generated **error** because Python tried to call the method with **3 arguments**



Solution



- The solution to this problem is a technique called **variable length arguments**

- In this technique , we define the function in such a way that it can accept any number of arguments from **0** to **infinite**

Syntax Of Variable Length Arguments



```
def <function_name>(* <arg_name>):
```

Function body

- As we can observe , to create a function with **variable length arguments** we simply prefix the argument name with an **asterisk**.
- **For example:**

```
def addnos(*a):
```

 - The function **addnos()** can now be called with as many **number of arguments** as we want and all the arguments will be stored inside the argument **a** which will be internally treated as **tuple**



Complete Example



```
def addnos(*a):
    sum = 0
    for x in a:
        sum = sum + x
    return sum
```

```
print(addnos(10,20))
print(addnos(10,20,30))
```

Output:

```
30
60
```



Exercise



- Write a function called **findlargest()** which accepts multiple strings as argument and returns the length of the largest string as well as the string itself



Solution



```
def findlargest(*names):
    max=0
    for s in names:
        if len(s)>max:
            max=len(s)
    return max
print(findlargest("January","February","March"))
```

Output:



Exercise



- Modify the previous example so that the function **findlargest()** now returns the largest string itself and not it's length



Solution

```
def findlargest(*names):
    max=0
    largest=""
    for s in names:
        if len(s)>max:
            max=len(s)
            largest=s
    return largest
print(findlargest("January","February","March"))
```

Output:

February



Point To Remember!



- A function cannot have 2 variable length arguments. So the following is wrong:

```
def addnos(*a,*b):
```



Point To Remember!



- If we have any other argument along with **variable length argument**, then it should be set **before** the **variable length argument**

```
def addnos(n,*a):  
    sum =n  
    for x in a:  
        sum=sum+x  
    return sum  
  
print(addnos(10,20,30))  
print(addnos(10,20,30,40))
```



Point To Remember!



- If we set the other argument used with **variable length argument** , **after** the **variable length argument** then:
 - While calling it we must pass it as **keyword argument** OR
 - Either it should be set as **default argument**

```
def addnos(*a,n):  
    sum =n  
    for x in a:  
        sum=sum+x  
    return sum  
  
print(addnos(20,30,n=0))  
print(addnos(20,30,40,n=0))
```

```
def addnos(*a,n=0):  
    sum =n  
    for x in a:  
        sum=sum+x  
    return sum  
  
print(addnos(20,30))  
print(addnos(20,30,40))
```



Guess The Output



```
def addnos(*a,n):  
    sum =n  
    for x in a:  
        sum=sum+x  
    return sum  
print(addnos(20,n=10,30))
```

Output:

SyntaxError: Positional argument follows keyword argument



Guess The Output



```
def show(a,b,c=3,d=4):  
    print(a,b,c,d)
```

show(10,20)

Output:

10 20 3 4



Guess The Output



```
def show(a,b,c=3,d=4):  
    print(a,b,c,d)
```

show(10,20,30,40)

Output:

10 20 30 40



Guess The Output



```
def show(a,b,c=3,d=4):  
    print(a,b,c,d)
```

```
show(d=10,a=20,b=30)
```

Output:

20 30 3 10



Guess The Output



```
def show(a,b,c=3,d=4):  
    print(a,b,c,d)
```

show()

Output:

TypeError



Guess The Output



```
def show(a,b,c=3,d=4):  
    print(a,b,c,d)
```

```
show(c=30,d=40,10,20)
```

Output:

SyntaxError



Guess The Output



```
def show(a,b,c=3,d=4):  
    print(a,b,c,d)
```

```
show(30,40,b=15)
```

Output:

TypeError : got multiple values for argument 'b'



PYTHON

LECTURE 21



Today's Agenda



- **User Defined Functions-III**
 - Variable Scope
 - Local Scope
 - Global Scope
 - Argument Passing



Variable Scopes



- The **scope** of a variable refers to the places from where we can see or access a variable.

- In Python , there are 4 types of scopes:
 - **Local : Inside a function body**
 - **Enclosing: Inside an outer function's body . We will discuss it later**
 - **Global: At the module level**
 - **Built In: At the interpreter level**

- In short we pronounce it as **LEGB**



Global Variable



□ GLOBAL VARIABLE

- A variable which is defined in the main body of a file is called a ***global*** variable.

- It will be visible **throughout** the **file**



Local Variable



□ LOCAL VARIABLE

- A variable which is defined **inside a function** is **local** to that function.
- It is accessible **from the point** at which it is defined until the **end of the function**.
- It exists for as long as the function is executing.
- Even the **parameter** in the function definition behave like **local variables**
- **When we use the assignment operator (=) inside a function, its default behaviour is to create a new local variable – unless a variable with the same name is already defined in the local scope.**



Example

```
s = "I love Python"  
def f0:  
    print(s)  
f0
```

Output:

I love Python

Since the variable **s** is **global** , we can access it from anywhere in our code



Example

```
s = "I love Python"  
def f():  
    print(s)
```

Output:

Since we have not called the function `f()`, so the statement `print(s)` will never get a chance to run



Example

```
def f():  
    print(s)  
s = "I love Python"  
f()
```

Output:

I love Python

Even though the variable **s** has been declared after the function **f()** , still it is considered to be **global** and can be accessed from anywhere in our code



Example

```
def f():  
    print(s)  
f()  
s="I love Python"
```

Output:

NameError !

Since we have called
the function **f()** ,
before declaring
variable **s** , so we get
NameError!



Example

```
def f():  
    s="I love Python"  
    print(s)  
f()
```

Output:

I love Python

The variable **s** now becomes a **local variable** and a **function** can easily access all the **local variables** inside it's definition



Example

```
def f0:  
    s="I love Python"  
    print(s)
```

f0

print(s)

Output:

I love Python

NameError!

The variable **s** is **local**
and cannot be
accessed from outside
it's function's
definition



Example

```
s="I love Python"  
def f():  
    s="I love C"  
    print(s)  
f()  
print(s)
```

Output:

I love C
I love Python

If a variable with
same name is defined
inside the scope of
function as well then
Python creates a **new
variable** in **local scope**
of the function and
uses it



Example

What if we want to use the same global variable inside the function also ?

```
s="I love Python"
```

```
def f():
```

```
    global s
```

```
    s="I love C"
```

```
    print(s)
```

```
f()
```

```
print(s)
```

Output:

I love C

I love C

To do this , we need a special keyword in Python called **global**. This keyword tells Python , not to create any new variable , rather use the variable from **global scope**



Guess The Output ?

```
s="I love Python"  
def f():  
    print(s)  
    s="I love C"  
    print(s)  
f()  
print(s)
```

Output:

UnboundLocalError:
Local variable s referenced before assignment

Now , this is a special case! .
In Python any variable
which is changed or created
inside of a function is **local**,
if it hasn't been declared as
a **global** variable. To tell
Python, that we want to use
the **global** variable, we have
to explicitly state this by
using the keyword "**global**"



Guess The Output ?

```
s="I love Python"  
def f():  
    global s  
    print(s)  
    s="I love C"  
    print(s)  
f()  
print(s)
```

Output:

I love Python
I love C
I love C



Guess The Output ?

```
a=1  
def f():  
    print ('Inside f() : ', a)  
def g():  
    a = 2  
    print ('Inside g() : ',a)  
def h():  
    global a  
    a = 3  
    print ('Inside h() : ',a)  
  
print ('global : ',a)  
f()  
print ('global : ',a)  
g()  
print ('global : ',a)  
h()  
print ('global : ',a)
```

Output:

global : 1
inside f():1
global: 1
inside g(): 2
global : 1
inside h(): 3
global : 3



Guess The Output ?

```
a=0  
if a == 0:  
    b = 1  
def my_function(c):  
    d = 3  
    print(c)  
    print(d)  
my_function(7)  
print(a)  
print(b)  
print(c)  
print(d)
```

Output:

7
3
0
1
NameError!



Guess The Output ?

```
def foo(x, y):  
    global a  
    a = 42  
    x,y = y,x  
    b = 33  
    b = 17  
    c = 100  
    print(a,b,x,y)
```

```
a, b, x, y = 1, 15, 3,4  
foo(17, 4)  
print(a, b, x, y)
```

Output:

42 17 4 17
42 15 3 4



Argument Passing



- There are **two** ways to pass arguments/parameters to function calls in **C programming**:
 - **Call by value**
 - **Call by reference.**



Call By Value



- In **Call by value**, original value is not modified.

- In **Call by value**, the value being passed to the function is locally stored by the function parameter as **formal argument**

- So , if we change the value of **formal argument**, it is changed for the **current function** only.

- These changes are not reflected in the **actual argument's** value



Call By Reference



- In **Call by reference** , the location (address) of **actual argument** is passed to **formal arguments**, hence any change made to formal arguments will also reflect in actual arguments.

- In **Call by reference, original value is modified** because we pass reference (address).



What About Python ?



- When asked whether **Python** function calling model is "**call-by-value**" or "**call-by-reference**", the correct answer is: **neither**.

- What Python uses , is actually called "**call-by-object-reference**"



A Quick Recap Of Variables



- We know that everything in **Python** is an **object**.
 - All **numbers** , **strings** , **lists** , **tuples** etc in **Python** are objects.
 - Now , recall , what happens when we write the following statement in **Python**:
- x=10
- An **object** is created in **heap** , storing the value **10** and **x** becomes the reference to that **object**.



A Quick Recap Of Variables



- Also we must recall that in **Python** we have 2 types of data : **mutable** and **immutable**.

- **Immutable types** are those which do not allow modification in object's data and examples are **int** , **float** , **string** ,**tuple** etc

- **Mutable types** are those which allow us to modify object's data and examples are **list** and **dictionary**

What Is Call By Object Reference ?



- Now , when we pass **immutable** arguments like **integers**, **strings** or **tuples** to a function, the passing acts like **call-by-value**.

- The ***object reference is passed*** to the function parameters.

- They can't be changed within the function, because they can't be changed at all, i.e. they are **immutable**.

What Is Call By Object Reference ?



- It's different, if we pass **mutable arguments**.
- They are also **passed by object reference**, but they can be **changed in place** in the function.
- If we pass a **list** to a function, elements of that **list** can be changed in place, i.e. the **list** will be changed even in the caller's scope.



Guess The Output ?

```
def show(a):
```

```
    print("Inside show , a is",a," It's id is",id(a))
```

```
a=10
```

```
print("Outside show, a i  
show(a)
```

Output:

Since Python uses Pass by object reference , so when we passed a , Python passed the address of the object pointed by a and this address was received by the formal variable a in the function's argument list. So both the references are pointing to the same object

```
Outside show, a is 10  It's id is 8791162737984  
Inside show , a is 10  It's id is 8791162737984
```



Guess The Output ?

```
def increment(a):  
    a=a+1  
  
a=10  
  
increment(a)  
  
print(a)
```

Output:
10

When we pass **a** to **increment(a)**, the function has the local variable **a** referring to the same object. Since integer is **immutable**, so Python is not able to **modify** the object's value to **11** in place and thus it created a new object. But the original variable **a** is still referring to the **same object** with the value **10**



Guess The Output ?

```
def show(mynumbers):  
    print("Inside show , mynumbers is",mynumbers)  
    mynumbers.append(40)  
    print("Inside show , mynumbe  
mynumbers=[10,20,30]  
print("Before calling show, n  
show(mynumbers)  
print("After calling show, my
```

Since **list** is a **mutable type** ,
so any change made in the
formal reference
mynumbers does not
create a new object in
memory . Rather it changes
the data stored in original
list

Output:

```
Before calling show, mynumbers is [10, 20, 30]  
Inside show , mynumbers is [10, 20, 30]  
Inside show , mynumbers is [10, 20, 30, 40]  
After calling show, mynumbers is [10, 20, 30, 40]
```



Guess The Output ?

```
def show(mynumbers):
    mynumbers=[50,60,70]
    print("Inside show , mynumbers is",mynumbers)
mynumbers=[10,20,30]
print("Before calling show, mynumbers is",mynumbers)
show(mynumbers)
print("After calling show, mynumbers is",mynumbers)
```

Output:

If we create a new object inside the function , then Python will make the formal reference mynumbers refer to that new object but the actual argument mynumbers , will still be referring to the actual object

```
Before calling show, mynumbers is [10, 20, 30]
Inside show , mynumbers is [50, 60, 70]
After calling show, mynumbers is [10, 20, 30]
```



Guess The Output ?



```
def foo(x):  
    x.append(3)  
    x = [8]  
    return x  
  
x=[1, 5]  
y= foo(x)  
print(x)  
print(y)
```

Output:

- [1,5,3]
- [8]



Guess The Output ?



```
def swap(a,b):  
    a,b=b,a  
  
a=10  
  
b=20  
  
swap(a,b)  
print(a)  
print(b)
```

Output:

10

20



Guess The Output ?



```
def changetoupper(s):
    s=s.upper()
s="bhopal"
changetoupper(s)
print(s)
```

Output:

bhopal



Guess The Output ?

```
def changetoupper(s):
    s=s.upper()
    return s
s="bhopal"
s=changetoupper(s)
print(s)
```

Output:

BHOPAL



PYTHON

LECTURE 22



Today's Agenda



- **User Defined Functions-IV**
 - Anonymous Functions OR Lambda Function

What Are Anonymous Functions ?



- An **anonymous** function is a function that is **defined without a name**.

- While **normal functions** are defined using the **def** keyword, we define **anonymous functions** using the **lambda** keyword.

- Hence, **anonymous functions** are also called **lambda** functions.



Syntax Of Lambda Functions



□ Syntax:

lambda [arg1,arg2,...]:[expression]

- **lambda** is a keyword/operator and can have any number of arguments.
- But it can have only one **expression**.
- Python evaluates the **expression** and returns the result automatically.



What Is An Expression ?



- An **expression** here is anything that can return some value.
- The following items qualify as expressions.
 - **Arithmetic operations** like $a+b$ and $a^{**}b$
 - **Function calls** like `sum(a,b)`
 - **A print statement** like `print("Hello")`

So, What Can Be Written In Lambda Expression ?



- **Assignment statements cannot be used in lambda** , because they don't return anything, not even **None** (null).
- Simple things such as **mathematical operations, string operations** etc. are OK in a lambda.
- **Function calls** are expressions, so it's OK to put a function call in a lambda, and to pass arguments to that function.
- Even **functions** that return **None**, like the **print** function in Python 3, can be used in a lambda.
- **Single line if – else** is also allowed as it also evaluates the condition and returns the result of **true** or **false** expression

How To Create Lambda Functions ?



- Suppose, we want to make a **function** which will ***calculate sum of two numbers.***
- In **normal approach** we will do as shown below:

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

- In case of **lambda function** we will write it as:

```
lambda a,b: a+b
```

Why To Create Lambda Functions ?



- A very common doubt is that when we can define our functions using **def** keyword , then **why we require lambda functions ?**

- The most common use for **lambda functions** is in code that requires **a simple one-line function**, where it would be an overkill to write a complete **normal function**.

- We will explore it in more detail when we will discuss **two** very important functions in **Python** called **map()** and **filter()**

How To Use Lambda Functions ?



- There are 2 ways to use a **Lambda Function**.
 - **Using it anonymously in inline mode**
 - **Using it by assigning it to a variable**

How To Use Lambda Functions ?



- Using it as **anonymous function**

```
print((lambda a,b: a+b)(2,3))
```

Output:

5

How To Use Lambda Functions ?



- Using it by assigning it to a variable

```
sum=lambda a,b: a+b
```

```
print(sum(2,3))  
print(sum(5,9))
```

Output:

```
5  
14
```

What is happening in this code ?

The statement **lambda a,b:a+b** , is creating a **FUNCTION OBJECT** and returning that object . The variable **sum** is referring to that object. Now when we write **sum(2,3)**, it behaves like function call



Guess The Output ?

```
sum=lambda a,b: a+b
```

```
print(type(sum))  
print(sum)
```

Since functions also are objects in Python , so they have their a unique memory address as well as their corresponding class as **function**

Output:

```
<class 'function'>  
<function <lambda> at 0x000000000050C1E0>
```



Example



```
squareit=lambda a: a*a
```

```
print(squareit(25))  
print(squareit(10))
```

Output:

```
625  
100
```



Example



```
import math  
sqrt=lambda a: math.sqrt(a)
```

```
print(sqrt(25))  
print(sqrt(10))
```

Output:

```
5.0  
3.1622776601683795
```



Exercise



- Write a lambda function that returns the first character of the string passed to it as argument

Solution:

```
firstchar=lambda str: str[0]
```

```
print("First character of Bhopal :",firstchar("Bhopal"))
print("First character of Sachin :",firstchar("Sachin"))
```

Output:

```
First character of Bhopal : B
First character of Sachin : S
```



Exercise



- Write a lambda function that returns the last character of the string passed to it as argument

Solution:

```
lastchar=lambda str: str[-1]
```

```
print("Last character of Bhopal :",lastchar("Bhopal"))
print("Last character of Sachin :",lastchar("Sachin"))
```

Output:

```
Last character of Bhopal : l
Last character of Sachin : n
```

Exercise



- Write a lambda function that returns True or False depending on whether the number passed to it as argument is even or odd

Solution:

```
iseven=lambda n: n%2==0  
print("10 is even :",iseven(10))  
print("7 is even:",iseven(7))
```

Output:

```
10 is even : True  
7 is even: False
```



Exercise



- Write a lambda function that accepts 2 arguments and returns the greater amongst them

Solution:

```
maxnum=lambda a,b: a if a>b else b  
print("max amongst 10 and 20 :",maxnum(10,20))  
print("max amongst 15 and 5 :",maxnum(15,5))
```

Output:

```
max amongst 10 and 20 : 20  
max amongst 15 and 5 : 15
```



PYTHON

LECTURE 23



Today's Agenda



• **User Defined Functions V**

- The map() Function
- The filter() Function
- Using map() and filter() with Lambda Expressions



What Is map() Function?



- As we have mentioned earlier, the advantage of the lambda operator can be seen when it is used in combination with the **map()** function.
- **map()** is a function which takes two arguments:
 - **r = map(func, iterable)**
- The first argument **func** is the **name of a function** and the second argument , **iterable** ,should be a **sequence** (e.g. a list , tuple ,string etc) or anything that can be used with **for** loop.
- **map()** applies the function **func** to all the elements of the sequence **iterable**



What Is map() Function?



- To understand this , let's solve a problem.

- Suppose we want to define a function called **square()** that can accept a number as argument and returns it's square.

- The definition of this function would be :

```
def square(num):  
    return num**2
```



What Is map() Function?



- Now suppose we want to call this function for the following list of numbers:
 - **mynums=[1,2,3,4,5]**
- One way to do this , will be to use a **for** loop

```
mynums=[1,2,3,4,5]
for x in mynums:
    print(square(x))
```



Complete Code

```
def square(num):  
    return num**2
```

```
mynums=[1,2,3,4,5]  
for x in mynums:  
    print(square(x))
```

Output:

1
4
9
16
25



Using map() Function



- Another way to solve the previous problem is to use the **map()** function .

- The **map()** function will accept 2 arguments from us.
 - The **first** argument will be the **name of the function square**

 - The **second** argument will be **the list mynums.**

- It will then apply the function **square** on every element of **mynum** and return the corresponding result as **map** object



Previous Code Using map()



```
def square(num):  
    return num**2
```

```
mynums=[1,2,3,4,5]  
result=map(square,mynums)  
print(result)
```

Output:

```
<map object at 0x0000000029030F0>
```

- As we can observe , the return value of **map()** function is a **map object**
- To convert it into actual numbers we can pass it to the **function list()**



Previous Code Using map()

```
def square(num):  
    return num**2
```

```
mynums=[1,2,3,4,5]  
result=map(square,mynums)  
sqrnum=list(result)  
print(sqrnum)
```

Output:

```
[1, 4, 9, 16, 25]
```

```
def square(num):  
    return num**2
```

```
mynums=[1,2,3,4,5] # we can club the 2 lines in 1 line  
sqrnum=list(map(square,mynums))  
print(sqrnum)
```



Previous Code Using map()



To make it even shorter we can directly pass the **list()** function to the function **print()**

```
def square(num):
    return num**2
```

```
mynums=[1,2,3,4,5]
print(list(map(square,mynums)))
```

Output:

```
[1, 4, 9, 16, 25]
```



Previous Code Using map()



In case we want to **iterate** over this **list**, then we can use **for loop**

```
def square(num):
```

```
    return num**2
```

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

```
for x in map(square,mynums):
```

```
    print(x)
```

Output:

```
1  
4  
9  
16  
25
```



Exercise



- Write a function called **inspect()** that accepts a string as argument and returns the word **EVEN** if the string is of **even length** and returns it's **first character** if the string is of **odd length**

Now call this function for first 3 month names



Solution

```
def inspect(mystring):
    if len(mystring)%2==0:
        return "EVEN"
    else:
        return mystring[0]
```

```
months=["January","February","March"]
print(list(map(inspect,months)))
```

Output:

```
['J', 'EVEN', 'M']
```



What Is filter() Function?



□ Like **map()** , **filter()** is also a function that is very commonly used in **Python** .

□ The function **filter ()** takes 2 arguments:

filter(function, sequence)

- The **first argument** should be a **function** which must return a **boolean value**
 - The **second argument** should be a **sequence** of **items**.
-
- Now the function **filter()** applies the function passed as argument to every **item** of the **sequence** passed as second argument.
 - If the function returned **True** for that item , **filter()** returns that **item** as part of it's return value otherwise the **item** is **not returned**.



What Is filter() Function?



- To understand this , let's solve a problem.
- Suppose we want to define a function called **check_even()** that can accept a **number** as argument and return **True** if it is even , otherwise it should return **False**
- The definition of this function would be :

```
def check_even(num):  
    return num%2==0
```



What Is filter() Function?



- Now suppose we have a list of numbers and we want to extract only even numbers from this list
 - **mynums=[1,2,3,4,5,6]**
- One way to do this , will be to use a **for** loop

```
mynums=[1,2,3,4,5,6]
for x in mynums:
    if check_even(x):
        print(x)
```



Complete Code

```
def check_even(num):  
    return num%2==0
```

```
mynums=[1,2,3,4,5,6]  
for x in mynums:  
    if check_even(x):  
        print(x)
```

Output:

2

4

6



Using filter() Function



- Another way to solve the previous problem is to use the **filter()** function .
- The **filter()** function will accept 2 arguments from us.
 - The **first** argument will be the **name of the function check_even**
 - The **second** argument will be **the list mynums**.
- It will then apply the function **check_even** on every element of **mynum** and if **check even** returned **True** for that element then **filter()** will return that element as a part of it's return value otherwise that element will not be returned



Previous Code Using filter()



```
def check_even(num):  
    return num%2==0  
  
mynums=[1,2,3,4,5,6]  
print(filter(check_even,mynums))
```

Output:

```
<filter object at 0x00000000029F3F60>
```

- As we can observe , the return value of **filter()** function is a **filter object**
- To convert it into actual numbers we can pass it to the **function list()**



Previous Code Using filter()



```
def check_even(num):  
    return num%2==0  
  
mynums=[1,2,3,4,5,6]  
print(list(filter(check_even,mynums)))
```

Output:

```
[2, 4, 6]
```



Previous Code Using filter()



In case we want to **iterate** over this **list**, then we can use **for loop** as shown below:

```
def check_even(num):  
    return num%2==0
```

```
mynums=[1,2,3,4,5,6]  
for x in filter(check_even,mynums):  
    print(x)
```

Output:

```
2  
4  
6
```



Guess The Output

```
def f1(num):  
    return num*num  
  
mynums=[1,2,3,4,5]  
print(list(filter(f1,mynums)))
```

Output:

[1,2,3,4,5]

Ideally , the function passed to **filter()** should return a **boolean** value. But if it doesn't return boolean value , then whatever value it returns **Python converts it to boolean** . In our case for each value in **mynums** the return value will be it's square which is a non-zero value and thus assumed to be **True**. So all the elements are returned by **filter()**



Guess The Output

```
def f1(num):  
    return num%2  
  
mynums=[1,2,3,4,5]  
print(list(filter(f1,mynums)))
```

Output:
[1,3,5]

For every **even number** the return value of the function **f1()** will be **0** which is assumed to be **False** and for every **odd number** the return value will be **1** which is assumed to be **True**. Thus **filter()** returns only those numbers for which **f1()** has returned **1**.



Guess The Output

```
def f1(num):  
    print("Hello")
```

```
mynums=[1,2,3,4,5]  
print(list(filter(f1,mynums)))
```

Output:

Hello
Hello
Hello
Hello
Hello
[]

Hello is displayed 5 times because the filter() function has called f1() function 5 times. Now for each value in mynums , since f1() has not returned any value , by default it's return value is assumed to be None which is a representation of False. Thus filter() returned an empty list.



Guess The Output

```
def f1(num):  
    pass
```

```
mynums=[1,2,3,4,5]  
print(list(filter(f1,mynums)))
```

Output:

```
[ ]
```

For each value in **mynums** , since **f1()** has not returned any value , by default it's return value is assumed to be **None** which is a representation of **False**. Thus **filter()** returned an empty list.



Guess The Output

```
def f1():  
    pass  
  
mynums=[1,2,3,4,5]  
print(list(filter(f1,mynums)))
```

The function **filter()** is trying to call **f1()** for every value in the list **mynums**. But since **f1()** is a **non-parametrized function** , this call generates **TypeError**

Output:

TypeError: f1() takes 0 positional arguments but 1 was given



Guess The Output



```
def f1():  
    pass  
  
mynums=[]  
print(list(filter(f1,mynums)))
```

Output:

[]



Guess The Output

```
def f1(num):  
    return num%2  
  
mynums=[1,2,3,4,5]  
print(list(map(f1,mynums)))
```

Output:

[1,0,1,0,1]

For every **even number** the return value of the function **f1()** will be **0** and for every **odd number** the return value will be **1**. Thus **map()** has returned a list containing **1** and **0** for each number in **mynums** based upon even and odd.



Guess The Output

```
def f1(num):  
    pass  
  
mynums=[1,2,3,4,5]  
print(list(map(f1,mynums)))
```

Since **f1()** is not returning anything , so it's return value by default is assumed to be **None** and because **map()** has internally called **f1()** 5 times , so the list returned contains **None** 5 times

Output:

[**None**,**None**,**None**,**None** ,**None**]



Guess The Output



```
def f1():  
    pass  
  
mynums=[]  
print(list(map(f1,mynums)))
```

Output:

[]

Using Lambda Expression With map() And filter()



- The best use of **Lambda Expression** is to use it with **map()** and **filter()** functions

- Recall that the keyword **lambda** creates an **anonymous function** and returns it's **address**.

Using Lambda Expression With map() And filter()



- So , we can pass this **lambda expression** as first argument to **map()** and **filter()** functions , since their first argument is the a **function object reference**
- In this way , we wouldn't be required to specially create a separate function using the keyword **def**



Using Lambdas With map()

```
def square(num):  
    return num**2
```

```
mynums=[1,2,3,4,5]  
sqrnum=list(map(square,mynums))  
print(sqrnum)
```



To convert the above code using **lambda**, we have to do 2 changes:

1. Remove the function **square()**
1. Rewrite this function as **lambda** in place of **first argument** while calling the function **map()**

Following will be the resultant code:

```
mynums=[1,2,3,4,5]  
sqrnum=list(map(lambda num: num*num,mynums))  
print(sqrnum)
```



Exercise



- Write a **lambda expression** that accepts a string as argument and returns it's **first character**

Now use this lambda expression in `map()` function to work on first 3 month names



Solution



```
months=["January","February","March"]
print(list(map(lambda mystring: mystring[0],months)))
```

Output:

```
['J', 'F', 'M']
```



Exercise



- Write a **lambda expression** that accepts a string as argument and returns the word **EVEN** if the string is of **even length** and returns it's **first character** if the string is of **odd length**

Now use this lambda expression in **map()** function to work on for first 3 month names



Solution



```
months=["January","February","March"]
print(list(map(lambda mystring: "EVEN" if len(mystring)%2==0 else
mystring[0],months)))
```

Output:

```
['J', 'EVEN', 'M']
```



Using Lambdas With filter()

```
def check_even(num):  
    return num%2==0
```

```
mynums=[1,2,3,4,5,6]  
print(list(filter(check_even,mynums)))
```

To convert the above code using **lambda** ,we have to same 2 steps as before.

Following will be the resultant code:

```
mynums=[1,2,3,4,5,6]  
print(list(filter(lambda num:num%2==0 ,mynums)))
```



Exercise



- Write a lambda expression that accepts a **character** as argument and returns **True** if it is a vowel otherwise **False**

Now ask the user to input his/her name and display only the vowels in the name . In case the name does not contain any vowel display the message **No vowels in your name**



Solution



```
name=input("Enter your name:")
vowels=list(filter(lambda ch: ch in "aeiou" ,name))
if len(vowels)==0:
    print("No vowels in your name")
else:
    print("Vowels in your name are:",vowels)
```

Output:

```
Enter your name:sachin
Vowels in your name are: ['a', 'i']
```



PYTHON

LECTURE 24



Today's Agenda



- **List -I**

- What Is A List ?
- Creating A List
- Accessing The List Elements
- Adding New Data In The List
- The Slice Operator With List



What Is A List ?



- Unlike **C++** or **Java**, **Python** doesn't has **arrays**.

- So , to hold a **sequence of values**, Python provides us a special built in class called '**list**' .

- Thus a **list** in **Python** is defined as ***a collection of values***.

Important Characteristics Of A List



□ The important characteristics of **Python lists** are as follows:

- **Lists are ordered.**
- **Lists can contain any arbitrary objects.**
- **List elements can be accessed by index.**
- **Lists can be nested to arbitrary depth.**
- **Lists are mutable.**
- **Lists are dynamic.**



How To Create A List ?



- In **Python**, a list is created by placing all the items (elements) inside a square bracket **[]**, separated by **commas**.
- It can contain **heterogeneous** elements also.

```
# empty list
```

```
my_list = []
```

```
# list of integers
```

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

```
# list with mixed datatypes
```

```
my_list = [1, "Hello", 3.4]
```



Other Ways Of Creating List



- We also can create a list by using the **list()** function

```
# Create an empty list
```

```
list1 = list()
```

```
# Create a list with elements 22, 31, 61
```

```
list2 = list([22, 31, 61])
```

```
# Create a list with strings
```

```
list3 = list(["tom", "jerry", "spyke"])
```

```
# Create a list with characters p, y, t, h, o, n
```

```
list4 = list("python")
```



Printing The List



- We can print a list in **three** ways:
 - Directly passing it to the **print()** function
 - Accessing individual elements using **subscript operator []**
 - Accessing multiple elements using **slice operator [:]**



Printing The List



```
mynums=[10,20,30,40,50]
print(mynums)
```

Output:

```
[10,20,30,40,50]
```



Accessing Individual Elements



- A list in Python has indexes running from **0** to **size-1**

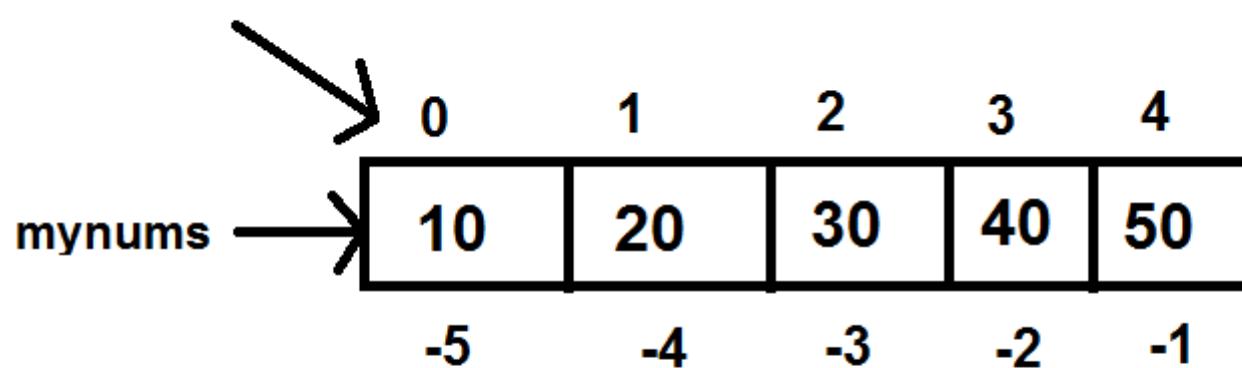
- **For example:**

- **mynums=[10,20,30,40,50]**
 - The above code will create a logical diagram in memory, where positive indexing will go from **0** to **4** and negative indexing from **-1** to **-5**



Accessing Individual Elements

Forward Indexing



Backward Indexing



Accessing Individual Elements



```
mynums=[10,20,30,40,50]
print(mynums[0])
print(mynums[1])
print(mynums[-3])
print(mynums[-2])
```

Output:

10

20

30

40

Accessing List Elements Using While Loop



```
mynums=[10,20,30,40,50]  
n=len(mynums)  
i=0  
while i<n:  
    print(mynums[i])  
    i=i+1
```

Output:

10
20
30
40
50

Just like **len()** works with **strings**, similarly it also works with **list** also and returns **number of elements in the list**

Accessing List Elements Using For Loop



```
mynums=[10,20,30,40,50]
```

```
for x in mynums:
```

```
    print(x)
```

Output:

10

20

30

40

50

Since list is a sequence type , so for loop can iterate over individual elements of the list



Exercise



- Redesign the previous code using for loop only to traverse the list in reverse order. Don't use slice operator



Solution



```
mynums=[10,20,30,40,50]
n=len(mynums)
for i in range(n-1,-1,-1):
    print(mynums[i])
```

Output:

50
40
30
20
10



Adding New Data In The List



- The most common way of adding a new element to an existing list is by calling the **append()** method.
- This method takes one argument and adds it at the end of the list

```
mynums=[10,20,30,40,50]
```

```
print(mynums)
```

```
mynums.append(60)
```

```
print(mynums)
```

Output:

```
[10,20,30,40,50]
```

```
[10,20,30,40,50,60]
```

Remember , lists are **mutable** . So **append()** method doesn't create a new list , rather it simply adds a new element to the existing list.

CAN YOU PROVE THIS ?



Solution

```
mynums=[10,20,30,40,50]
print(mynums)
print(id(mynums))
mynums.append(60)
print(mynums)
print(id(mynums))
```

Output:

```
[10, 20, 30, 40, 50]
35676744
[10, 20, 30, 40, 50, 60]
35676744
```

As we can see in the both the cases the **id()** function is returning the **same address** . This means that no new list was created.



Exercise



- Write a program to accept five integers from the user , store them in a list . Display these integers and also display their sum

- Output:

```
Enter 1 element:10
Enter 2 element:20
Enter 3 element:30
Enter 4 element:40
Enter 5 element:50
The list is:
10
20
30
40
50
Sum is 150
```



Solution



```
mynums=[]
i=1
while i<=5:
    x=int(input("Enter "+str(i)+" element:"))
    mynums.append(x)
    i=i+1
print("The list is:")
sum=0
for x in mynums:
    print(x)
    sum+=x
print("Sum is",sum)
```



Slice Operator With List



- Just like we can apply slice operator with strings , similarly Python allows us to apply slice operator with lists also.

- **Syntax:** `list_var[x:y]`
 - **x** denotes the **start index** of slicing and **y** denotes the **end index** . But **Python** ends slicing at **y-1** index.



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[1:4])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[3:5])
```

- **Output:**

[20,30,40]

- **Output:**

[40,50]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[0:4])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[0:10])
```

- **Output:**

[10,20,30,40]

- **Output:**

[10,20,3040,50]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[2:2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[6:10])
```

- **Output:**

[]

- **Output:**

[]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[1: ])
```

- **Output:**

[20,30,40,50]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[:3])
```

- **Output:**

[10,20,30]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[ :-2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-2:])
```

- **Output:**

[10, 20,30]

- **Output:**

[40,50]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-2:1])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-2:-2])
```

- **Output:**

[]

- **Output:**

[]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-2:2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-2:4])
```

- **Output:**

[]

- **Output:**

[40]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-4:2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[1:-2])
```

- **Output:**
[20]

- **Output:**
[20,30]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-2: -1])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1:-2])
```

- **Output:**

[40]

- **Output:**

[]



Using Step Value

- String slicing can accept a **third parameter** also after the two index numbers.
- The **third parameter** is called **step value**.
- So the complete syntax of slicing operator is:

s[begin:end:step]

- Step value indicates *how many characters to move forward after the first character is retrieved* from the string and it's default value is **1** , but can be changed as per our choice.



Using Step Value

- Another point to understand is that if **step** is **positive** or **not mentioned** then
 - **Movement is in forward direction (L → R)**
 - **Default for start is 0 and end is len**
- But if **step** is **negative** , then
 - **Movement is in backward direction (R ← L)**
 - **Default for start is -1 and end is -(len+1)**



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[1:4:2])
```

- **Output:**
[20,40]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[1:4:0])
```

- **Output:**
**ValueError: Slice
step cannot be
0**



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[4:1:-1])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[4:1:-1])
```

- **Output:**

[]

- **Output:**

[50,40,30]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[:])
```

- **Output:**

[10,20,30,40,50]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[::-1])
```

- **Output:**

[50,40,30,20,10]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[::-2])
```

- **Output:**

[50,30,10]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[::-2])
```

- **Output:**

[10,30,50]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-1:-4:])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1:-4:-1])
```

- **Output:**

```
[]
```

- **Output:**

```
[50,40,30]
```



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-4:-1])
```

- **Output:**

[20,30,40]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-4:-1:-1])
```

- **Output:**

[]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-1: :-2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1::2])
```

- **Output:**

[50,30,10]

- **Output:**

[50]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-1:4 :2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-3::-1])
```

- **Output:**

[]

- **Output:**

[30,20,10]



The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[:-3:-1])
```

- **Output:**
[50,40]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1:-1:-1])
```

- **Output:**
[]



PYTHON

LECTURE 25



Today's Agenda



• **List -II**

- Modifying A List
- Deletion In A List
- Appending / Prepending Items In A List
- Multiplying A List
- Membership Operators On List



Modifying A List

- Python allows us to **edit/change/modify** an element in a list by simply using it's **index** and assigning a **new value** to it

Syntax:

```
list_var[index_no]=new_value
```

Example

```
sports=["cricket","hockey","football"]
```

```
print(sports)
```

```
sports[1]="badminton"
```

```
print(sports)
```

Output

```
['cricket', 'hockey', 'football']
['cricket', 'badminton', 'football']
```



Guess The Output ?



```
sports=["cricket","hockey","football"]
print(sports)
sports[3]="badminton"
print(sports)
```

Output:

```
['cricket', 'hockey', 'football']
Traceback (most recent call last):
  File "listdemo9.py", line 3, in <module>
    sports[3]="badminton"
IndexError: list assignment index out of range
```



Modifying Multiple Values

- Python allows us to modify **multiple continuous list values** in **a single statement**, which is done using the regular **slice operator**.

Syntax:

`list_var[m:n]=[list of new value]`

Example:

```
sports=["cricket","hockey","football","snooker"]
```

```
print(sports)
```

```
sports[1:3]=["badminton","tennis"]
```

```
print(sports)
```

Output:

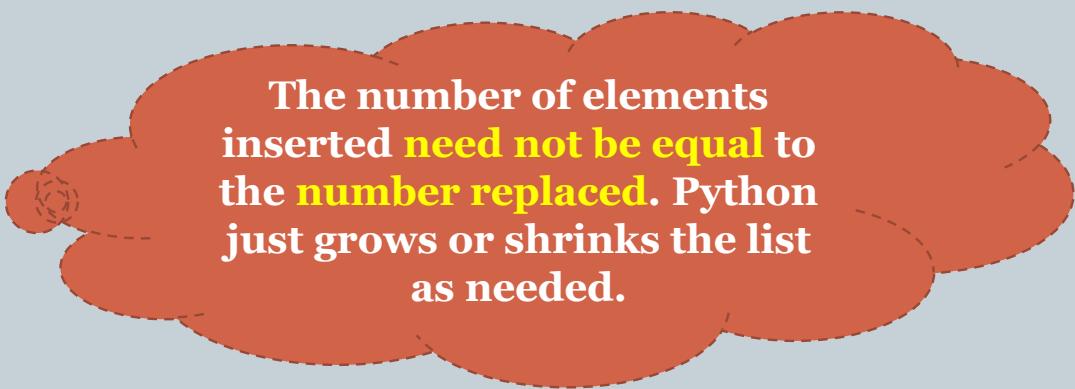
```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'snooker']
```



Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:3]=["badminton","tennis","rugby","table
tennis"]
print(sports)
```

Output:

A thought bubble graphic composed of three overlapping orange rounded rectangles. Inside the bubble, there is a small icon of a brain. The text within the bubble reads: "The number of elements inserted **need not be equal** to the **number replaced**. Python just grows or shrinks the list as needed."

The number of elements inserted **need not be equal** to the **number replaced**. Python just grows or shrinks the list as needed.

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'rugby', 'table tennis', 'snooker']
```



Guess The Output ?



```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:2]=["badminton","tennis","rugby","table
tennis"]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'rugby', 'table tennis', 'football', 'snooker']
[]
```



Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:1]=["badminton","tennis"]
print(sports)
```

Output:

If we have **end index same or less than start index** , then Python **doesn't remove anything** . Rather it simply **inserts new elements** at the given index and **shifts the existing element**

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'hockey', 'football', 'snooker']
```



Guess The Output ?



```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:0]=["badminton","tennis"]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'hockey', 'football', 'snooker']
```



Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:-1]=["badminton","tennis"]
print(sports)
```

Since **-1** is present in the list ,
Python **removed items** from **1** to
second last item of the list and
inserted new items there

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'snooker']
```



Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:-2]=["badminton","tennis"]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'football', 'snooker']
```



Deleting Item From The List



- **Python** allows us to delete an item from the list by calling the **operator/keyword** called **del**

Syntax:

del list_var[index_no]

Example:

```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[3]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'hockey', 'football']
```



Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[4]
print(sports)
```

Subscript operator will generate
IndexError whenever we pass
invalid index

Output:

```
['cricket', 'hockey', 'football', 'snooker']
Traceback (most recent call last):
  File "listdemo11.py", line 3, in <module>
    del sports[4]
IndexError: list assignment index out of range
```



Deleting Multiple Items



- Python allows us to **delete** multiple items from the list in **2 ways:**

- By assigning empty list to the appropriate slice

- OR

- By passing slice to the del operator



Example



□ Assigning Empty List

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:3]=[]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'snooker']
```



Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:5]=[]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket']
```

Slice operator never generates IndexError , so the code will work fine and remove all the items from given start index to the end of the list



Example

□ Passing slice to del operator

```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[1:3]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'snooker']
```



Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]  
print(sports)  
del sports[1:5]  
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']  
['cricket']
```

Here also , since we have used the
slice operator , no exception will
arise



Guess The Output ?



```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[0:4]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
[]
```



Deleting Entire List

- We can delete or remove the **entire list object** as well as it's **reference** from the memory by passing the **list object reference** to the **del** operator

Syntax:

`del list_var`

Example:

```
sports=["cricket","hockey","football","snooker"]
```

```
print(sports)
```

```
del sports
```

```
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'snooker']
Traceback (most recent call last):
  File "listdemo11.py", line 4, in <module>
    print(sports)
NameError: name 'sports' is not defined
```

Appending Or Prepending Items To A List



- **Additional items** can be added to the **start** or **end** of a list using the **+** concatenation operator or the **`+ =`** compound assignment operator
- The only condition is that the item to be concatenated must be a **list**

Example:

```
outdoor=["cricket","hockey","football"]
indoor=["carrom","chess","table-tennis"]
allsports=outdoor+indoor
print(allsports)
```

```
['cricket', 'hockey', 'football', 'carrom', 'chess', 'table-tennis']
```



Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=["carrom","chess","table-tennis"]+sports
print(sports)
```

Output:

```
['carrom', 'chess', 'table-tennis', 'cricket', 'hockey', 'football']
```



Guess The Output ?



evens=[4,6,8]

evens=2+evens

print(evens)

Output:

```
evens=2+evens
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```



Guess The Output ?



evens=[4,6,8]

evens=list(2)+evens

print(evens)

Output:

```
evens=list(2)+evens
```

```
TypeError: 'int' object is not iterable
```



Guess The Output ?



evens=[4,6,8]

evens=[2]+evens

print(evens)

Output:

```
[2, 4, 6, 8]
```



Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports+"boxing"
print(sports)
```

Output:

```
 sports=[cricket,hockey,football]
TypeError: can only concatenate list (not "str") to list
```



Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports+list("boxing")
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'b', 'o', 'x', 'i', 'n', 'g']
```



Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports+["boxing"]
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'boxing']
```



Multiplying A List



- **Python** allows us to **multiply a list by an integer** and when we do so it makes copies of list items that many number of times, while preserving the order.
- **Syntax:**

list_var * n

Example:

```
sports=["cricket","hockey","football"]
```

```
sports=sports*3
```

```
print(sports)
```

Output:

```
['cricket', 'hockey', 'football', 'cricket', 'hockey', 'football', 'cricket', 'hockey', 'football']
```



Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports*3.0
print(sports)
```

Output:

```
sports=sports*3.0
TypeError: can't multiply sequence by non-int of type 'float'
```



Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports*["boxing"]
print(sports)
```

Output:

```
sports=sports*["boxing"]
TypeError: can't multiply sequence by non-int of type 'list'
```



Membership Operator On List

- We can apply **membership operators in** and **not in** on the **list** to **search** for a particular **item**

Syntax:

element in list_var

Example:

```
sports=["cricket","hockey","football"]
```

```
print("cricket" in sports)
```

Output:

```
True
```

Exercise



- Write a program to accept **5 unique integers** from the user. Make sure if the integer being entered is **already present** in the list your code displays the message "**Item already present**" and ask the user to reenter the integer.

Output:

```
Enter 5 unique integers:  
Enter element:1  
Enter element:2  
Enter element:1  
Item already present  
Enter element:2  
Item already present  
Enter element:3  
Enter element:4  
Enter element:4  
Item already present  
Enter element:5  
integers inputted by you are:  
1  
2  
3  
4  
5
```



Solution



```
myints=[]
print("Enter 5 unique integers:")
i=0
while i<=4:
    item=int(input("Enter element:"))
    if item in myints:
        print("Item already present!")
        continue
    myints.append(item)
    i=i+1

print("integers inputted by you are:")
for x in myints:
    print(x)
```

Exercise



- Write a program to accept 2 lists from the user of 5 nos each . Assume each list will have unique nos
Now find out how many items in these lists are common .

Output:

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:2  
Enter element:3  
Enter element:4  
Enter element:5  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have 2 items common
```

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:3  
Enter element:5  
Enter element:7  
Enter element:9  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have no common items
```

Exercise



- Rewrite the previous code so that your code also displays the items which are common in both the lists

Output:

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:2  
Enter element:3  
Enter element:4  
Enter element:5  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have 2 items common  
These items are: [2, 4]
```

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:3  
Enter element:5  
Enter element:7  
Enter element:9  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have no common items
```



PYTHON

LECTURE 26



Today's Agenda



- **List -III**
 - Built In Functions For List



Built In Functions For List



- There are some **built-in functions** in **Python** that we can use on **lists**.
- These are:
 - **len()**
 - **max()**
 - **min()**
 - **sum()**
 - **sorted()**
 - **list()**
 - **any()**
 - **all()**



The **len()** Function

- Returns the **number of items** in the list

Example:

```
fruits=["apple","banana","orange",None]  
print(len(fruits))
```

Output:

4



The **max()** Function



- Returns the **greatest** item present in the list

Example:

```
nums=[5,2,11,3]
```

```
print(max(nums))
```

Output:

11



Guess The Output ?



```
months=["january","may","december"]  
print(max(months))
```

Output:

may



Guess The Output ?



```
booleans=[False,True]  
print(max(booleans))
```

Output:

True



Guess The Output ?



```
mynums=[1.1,1.4,0.9]  
print(max(mynums))
```

Output:

1.4



Guess The Output ?



```
mynums=[True,5,False]  
print(max(mynums))
```

Output:

5



Guess The Output ?



```
mynums=[0.2,0.4,True,0.5]  
print(max(mynums))
```

Output:

True



Guess The Output ?



```
mynums=["True",False]  
print(max(mynums))
```

Output:

```
print(max(mynums))  
TypeError: '>' not supported between instances of 'bool' and 'str'
```



Guess The Output ?



```
values=[10,"hello",20,"bye"]  
print(max(values))
```

Output:

```
print(max(values))  
TypeError: '>' not supported between instances of 'str' and 'int'
```



Guess The Output ?



```
fruits=["apple","banana","orange"]  
print(max(fruits))
```

Output:

orange



Guess The Output ?



```
fruits=["apple","banana","orange",None]  
print(max(fruits))
```

Output:

```
print(max(fruits))  
TypeError: '>' not supported between instances of 'NoneType' and 'str'
```



The **min()** Function



- Returns the **least** item present in the list

Example:

```
nums=[5,2,11,3]  
print(min(nums))
```

Output:

2



Guess The Output ?



```
months=["january","may","december"]  
print(min(months))
```

Output:

december



The **sum()** Function



- Returns the **sum** of all the **items** present in the list .
However items must be of Numeric or boolean type

Example:

```
nums=[10,20,30]  
print(sum(nums))
```

Output:

60



Guess The Output ?



```
nums=[10,20,30,True,False]  
print(sum(nums))
```

Output:

61



Guess The Output ?



```
nums=['1','2','3']  
print(sum(nums))
```

Output:

```
print(sum(nums))  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



Guess The Output ?



```
nums=[2.5,3.5,4.5]  
print(sum(nums))
```

Output:

10.5



The **sorted()** Function



- Returns a **sorted version** of the **list** passed as argument.

Example:

```
nums=[7,4,9,1]  
print(sorted(nums))  
print(nums)
```

Output:

```
[1, 4, 7, 9]  
[7, 4, 9, 1]
```



Guess The Output ?



```
months=["january","may","december"]  
print(sorted(months))
```

Output:

[“december”, “january”, “may”]



Guess The Output ?



```
months=["january","may","december",3]  
print(sorted(months))
```

Output:

```
print(sorted(months))  
TypeError: '<' not supported between instances of 'int' and 'str'
```



Guess The Output ?



```
values=[2.4,1.0,2,3.6]  
print(sorted(values))
```

Output:

[1.0,2,2.4,3.6]



Guess The Output ?



```
values=["bhupal","bhop","Bhopal"]  
print(sorted(values))
```

Output:

["Bhopal", "bhop", "bhupal"]



Sorting In Descending Order

- To **sort** the **list** in **descending order** , we can pass the **keyword argument reverse** with value set to **True** to the function **sorted()**

Example:

```
nums=[3,1,5,2]
```

```
print(sorted(nums,reverse=True))
```

Output:

```
[5, 3, 2, 1]
```



The **list()** Function

- The **list()** function converts an **iterable** i.e **tuple** , **range**, **set** , **dictionary** and **string** to a **list**.

Syntax:

list(iterable)

Example:

```
city="bhopal"  
x=list(city)  
print(x)
```

Output:

```
['b', 'h', 'o', 'p', 'a', 'l']
```



Guess The Output ?



n=20

x=list(n)

print(x)

Output:

```
x=list(n)
```

```
TypeError: 'int' object is not iterable
```



Guess The Output ?



```
n="20"
```

```
x=list(n)
```

```
print(x)
```

Output:

```
[ '2', '0' ]
```



Guess The Output ?

```
t=(10,20,30)
```

```
x=list(t)
```

```
print(x)
```

Output:

```
[10, 20, 30]
```

This is a
tuple



The **any()** Function

- The **any()** function accepts a **List** as argument and returns **True** if atleast **one element** of the **List** is **True**. If not, this method returns **False**. If the **List** is empty, then also it returns **False**

Syntax:

list(iterable)

Example:

```
x = [1, 3, 4, 0]
print(any(x))
```

Output:

True



Guess The Output ?



```
x = [0, False]  
print(any(x))
```

Output:

False



Guess The Output ?



```
x = [0, False, 5]  
print(any(x))
```

Output:

True



Guess The Output ?



x= []

print(any(x))

Output:

False



The **all()** Function



- The **all()** function accepts a **List** as argument and returns **True** if **all the elements** of the **List** are **True** or if the **List** is **empty**. If not, this method returns **False**.

Syntax:

all(iterable)

Example:

```
x = [1, 3, 4, 0]  
print(all(x))
```

Output:

False



Guess The Output ?



```
x = [0, False]  
print(all(x))
```

Output:

False



Guess The Output ?



```
x = [1,3,4,5]  
print(all(x))
```

Output:

True



Guess The Output ?



```
x = [0, False, 5]  
print(all(x))
```

Output:

False



Guess The Output ?



```
x= []
print(all(x))
```

Output:

True



PYTHON

LECTURE 27



Today's Agenda



- **List -IV**
 - Methods Of List



List Methods



- There are some **methods** also in **Python** that we can use on **lists**.
- These are:
 - **append()**
 - **extend()**
 - **insert()**
 - **index()**
 - **count()**
 - **remove()**
 - **pop()**
 - **clear()**
 - **sort()**
 - **reverse()**



The **append()** Method



- Adds a **single element** to the **end** of the list . Modifies the list in place but doesn't return anything

Syntax:

list_var.append(item)

Example:

primes=[2,3,5,7]

primes.append(11)

print(primes)

Output:

[2, 3, 5, 7, 11]



Guess The Output ?

```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.append(wild_animal)
print(animal)
```

Output:

```
['cat', 'dog', 'rabbit', ['tiger', 'fox']]
```

It's important to notice that, **append()** adds entire list as a single element .

If we need to add items of a list to the another list (rather than the list itself), then we must use the **extend()** method .



Guess The Output ?

```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.append(wild_animal)
print(animal[3])
print(animal[3][0])
print(animal[3][0][0])
print(animal[3][1][0])
```

Output:

```
['tiger', 'fox']
tiger
t
f
```



Exercise



- Write a program to accept an alphanumeric string from the user. Now extract only digits from the given input and add it to the list . Finally print the list as well as it's sum. **Make sure your list contains numeric representation of digits**

Output:

```
Enter an alphanumeric string:a1b2c345de56
[1, 2, 3, 4, 5, 5, 6]
```



Solution

```
text=input("Enter an alphanumeric string:")
nums=[]
for x in text:
    if x in "0123456789":
        nums.append(int(x))
print(nums)
```



The **extend()** Method



- **extend()** also adds to the **end of a list**, *but the argument is expected to be an iterable*.
- The items in **<iterable>** are added individually.
- Modifies the list in place but **doesn't return anything**

Syntax:

`list_var.extend(iterable)`

Output:

Example:

`primes=[2,3,5,7]`

`primes.extend([11,13,17])`

`print(primes)`

`[2, 3, 5, 7, 11, 13, 17]`



Guess The Output ?



```
primes=[2,3,5,7]
primes.extend(11)
print(primes)
```

Output:

```
primes.extend(11)
TypeError: 'int' object is not iterable
```



Guess The Output ?



```
primes=[2,3,5,7]  
primes.extend([11])  
print(primes)
```

Output:

```
[2, 3, 5, 7, 11]
```



Guess The Output ?

```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.extend(wild_animal)
print(animal)
```

Output:

```
['cat', 'dog', 'rabbit', 'tiger', 'fox']
```



Guess The Output ?



```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.extend(wild_animal)
print(animal[3])
print(animal[3][0])
print(animal[3][1])
```

Output:

```
tiger
t
i
```



Guess The Output ?



```
colors=["red","green"]
colors.extend("blue")
print(colors)
```

Output:

```
['red', 'green', 'b', 'l', 'u', 'e']
```



Guess The Output ?



```
colors=["red","green"]
colors.extend(["blue"])
print(colors)
```

Output:

```
['red', 'green', 'blue']
```



Guess The Output ?



```
a = [1, 2, 3]  
b = [4, 5, 6].extend(a)  
print(b)
```

Output:

None



The **insert()** Method



- The **insert()** method inserts the element to the list at the given index.
- Modifies the list in place but **doesn't return anything**

Syntax:

`list_var.insert(index,item)`

Example:

```
primes=[2,3,7,9]  
primes.insert(2,5)  
print(primes)
```

Output:

```
[2, 3, 5, 7, 9]
```



Guess The Output ?



```
primes=[2,3,7,9]  
primes.insert(-2,5)  
print(primes)
```

Output:

```
[2, 3, 5, 7, 9]
```



Guess The Output ?

```
primes=[2,3,5,7]  
primes.insert(5,9)  
print(primes)
```

Output:

```
[2, 3, 5, 7, 9]
```

The method **insert()** works like slicing operator . So even if the index given is beyond range , it will add the element at the end.



Guess The Output ?



```
primes=[2,3,5,7]
primes.insert(-5,9)
print(primes)
```

Output:

```
[9, 2, 3, 5, 7]
```



Exercise



- Write a program to accept any 5 random integers from the user and add them in a list in such a way that list always remains sorted. **DO NOT USE THE FUNCTION sort()**

Output:

```
Enter any 5 random integers:  
4  
1  
6  
2  
3  
Sorted list is:  
[1, 2, 3, 4, 6]
```



Solution

```
i=1
sortednums=[]
print("Enter any 5 random integers:")
while i<=5:
    n=int(input())
    pos=0
    for x in sortednums:
        if x>n:
            break
        pos=pos+1
    sortednums.insert(pos,n)
    i=i+1
print("Sorted list is:")
print(sortednums)
```



The **index()** Method



- The **index()** method searches an element in the **list** and returns it's **index**.
- If the element occurs **more than once** it returns it's **smallest/first position**.
- If element is **not found**, it raises a **ValueError** exception

Syntax:

list_var.index(item)

Example:

```
primes=[2,3,5,7]
pos=primes.index(5)
print("position of 5 is",pos)
```

Output:

```
position of 5 is 2
```



Guess The Output ?

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
pos = vowels.index('e')
print('The index of e:',pos)
pos = vowels.index('i')
print('The index of i:',pos)
```

Output:

```
The index of e: 1
The index of i: 2
```



Guess The Output ?



```
mynums = [10,20,30,40,50]
x = mynums.index(20)
print("20 occurs at",x,"position")
x = mynums.index(60)
print("60 occurs at",x,"position")
x = mynums.index(10)
print("10 occurs at",x,"position")
```

Output:

```
20 occurs at 1 position
Traceback (most recent call last):
  File "listdemo30.py", line 4, in <module>
    x = mynums.index(60)
ValueError: 60 is not in list
```



Guess The Output ?

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
pos = vowels.index('f')
print('The index of e:',pos)
pos = vowels.index('i')
print('The index of i:',pos)
```

Output:

```
Traceback (most recent call last):
  File "listdemo30.py", line 2, in <module>
    pos = vowels.index('f')
ValueError: 'f' is not in list
```



The **count()** Method



- The **count()** method returns the **number of occurrences** of an element in a **list**
- In simple terms, it **counts** how many times an element has occurred in a **list** and returns it.

Syntax:

list_var.count(item)

Output:

Example:

```
country=['i','n','d','i','a']
```

```
x=country.count('i')
```

```
print("i occurs",x,"times in",country)
```

```
i occurs 2 times in '['i', 'n', 'd', 'i', 'a']'
```



Guess The Output ?

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
x = vowels.count('i')
print("i occurs",x,"times")
x = vowels.count('e')
print("e occurs",x,"times")
x = vowels.count('j')
print("j occurs",x,"times")
```

Output:

```
i occurs 2 times
e occurs 1 times
j occurs 0 times
```



Guess The Output ?



```
points = [1, 4, 2, 9, 7, 8, 9, 3, 1]  
x = points.count(9)  
print("9 occurs",x,"times")
```

Output:

```
9 occurs 2 times
```



Guess The Output ?



```
strings = ['Cat', 'Bat', 'Sat', 'Cat', 'cat', 'Mat']
x=strings.count("Cat")
print("Cat occurs",x,"times")
```

Output:

```
Cat occurs 2 times
```



The **remove()** Method

- The **remove()** method **searches** for the given element in the list and **removes the first matching element**.
- If the **element**(argument) passed to the **remove()** method doesn't exist, **ValueError** exception is thrown.

Syntax:

list_var.remove(item)

Output:

Example:

```
vowels=['a','e','i','o','u']
vowels.remove('a')
print(vowels)
```

```
['e', 'i', 'o', 'u']
```



Guess The Output ?



```
subjects=["phy","chem","maths"]
subjects.remove("chem")
print(subjects)
```

Output:

```
['phy', 'maths']
```



Guess The Output ?

```
subjects=["phy","chem","maths","chem"]
subjects.remove("chem")
print(subjects)
subjects.remove("chem")
print(subjects)
subjects.remove("chem")
print(subjects)
```

Output:

```
['phy', 'maths', 'chem']
['phy', 'maths']
Traceback (most recent call last):
  File "listdemo33.py", line 6, in <module>
    subjects.remove("chem")
ValueError: list.remove(x): x not in list
```



The **pop()** Method



- The **pop()** method **removes** and **returns** the element at the given index (passed as an argument) from the list.

Syntax:

list_var.pop(index)

- Important points about **pop()** method:
 - If the **index** passed to the **pop()** method is not in the range, it throws **IndexError: pop index out of range** exception.
 - The **parameter** passed to the **pop()** method is **optional**. If no parameter is passed, the **default index -1 is passed** as an argument which **returns the last element**
 - The **pop()** method returns the element present at the given index.
 - Also, the **pop()** method updates the list after removing the element



Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(1))
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
banana
['apple', 'cherry']
```



Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop())
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
cherry
['apple', 'banana']
```



Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(3))
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
Traceback (most recent call last):
  File "listdemo34.py", line 3, in <module>
    print(fruits.pop(3))
IndexError: pop index out of range
```



Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(-3))
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
apple
['banana', 'cherry']
```



Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(-4))
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
Traceback (most recent call last):
  File "listdemo34.py", line 3, in <module>
    print(fruits.pop(-4))
IndexError: pop index out of range
```



del v/s pop() v/s remove()



- **pop()** : Takes Index , removes element & returns it
- **remove()** : Takes value, removes first occurrence and returns nothing
- **del** : Takes index, removes value at that index and returns nothing
- Even their exceptions are also different if index is wrong or element is not present:
 - **pop()** : throws **IndexError**: pop index out of range
 - **remove()**: throws **ValueError**: list.remove(x): x not in list
 - **del**: throws **IndexError**: list assignment index out of range



The **clear()** Method



- The **clear()** method removes all items from the list.
- It only empties the given **list** and doesn't return any value.

Syntax:

list_var.clear()

Example:

```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
fruits.clear()
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
[]
```



The **sort()** Method



- The **sort()** method **sorts** the elements of a given list.
- The order can be **ascending** or **descending**

Syntax:

`list_var.sort(reverse=True|False, key=name of func)`

Parameter Values:

Parameter Name	Description
reverse	Optional. <code>reverse=True</code> will sort the list descending. Default is <code>reverse=False</code>
key	Optional. A function to specify the sorting criteria(s)



Guess The Output ?



```
vowels = ['e', 'a', 'u', 'o', 'i']
```

```
vowels.sort()
```

```
print(vowels)
```

Output:

```
['a', 'e', 'i', 'o', 'u']
```



Guess The Output ?



```
vowels = ['e', 'a', 'u', 'o', 'i']
vowels.sort(reverse=True)
print(vowels)
```

Output:

```
['u', 'o', 'i', 'e', 'a']
```



Guess The Output ?



```
a = ["bee", "wasp", "moth", "ant"]  
a.sort()  
print(a)
```

Output:

```
['ant', 'bee', 'moth', 'wasp']
```



Guess The Output ?



```
a = ["bee", "wasp", "moth", "ant"]  
a.sort(reverse=True)  
print(a)
```

Output:

```
['wasp', 'moth', 'bee', 'ant']
```



Guess The Output ?



```
a = ["bee", "wasp", "moth", "ant"]  
a.sort(key=len)  
print(a)
```

Output:

```
['bee', 'ant', 'wasp', 'moth']
```



Guess The Output ?



```
a = ["january", "february", "march"]  
a.sort(key=len,reverse=True)  
print(a)
```

Output:

```
['february', 'january', 'march']
```



Guess The Output ?



```
mylist = ["a",10,True]  
mylist.sort()  
print(mylist)
```

Output:

```
Traceback (most recent call last):  
  File "listdemo34.py", line 2, in <module>  
    mylist.sort()  
TypeError: '<' not supported between instances of 'int' and 'str'
```



Guess The Output ?



```
mylist = [25,10,True,False]  
mylist.sort()  
print(mylist)
```

Output:

```
[False, True, 10, 25]
```

Passing Our Own Function As Key



- We also can pass our own function name to be used as key but it should take only 1 argument and return some value based on that argument.
- **This return value will be used by Python as key to sorting**



Guess The Output ?



```
def sortSecond(val):  
    return val[1]  
  
list1 = [(1,2),(3,3),(1,1)]  
list1.sort(key=sortSecond)  
print(list1)
```

Output:

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



Guess The Output ?



```
def sortSecond(val):  
    return 0  
  
list1 = [(1,2),(3,3),(1,1)]  
list1.sort(key=sortSecond)  
print(list1)
```

Output:

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



Guess The Output ?

```
student_rec = [['john', 'A', 12],['jane', 'B', 7],['dave',  
 'B', 10]]
```

```
student_rec.sort()  
print(student_rec)
```

Output:

```
[['dave', 'B', 10], ['jane', 'B', 7], ['john', 'A', 12]]
```



Guess The Output ?



```
def myFunc(val):  
    return val[2]
```

```
student_rec = [['john', 'A', 12],['jane', 'B', 7],['dave',  
 'B', 10]]
```

```
student_rec.sort(key=myFunc)  
print(student_rec)
```

Output:

```
[['jane', 'B', 7], ['dave', 'B', 10], ['john', 'A', 12]]
```



The **reverse()** Method



- The **reverse()** method **reverses** the elements of a given list.
- It doesn't return any value. It only **reverses the elements** and **updates the list**.

Syntax:

`list_var.reverse()`

Example:

```
os = ['Windows', 'macOS', 'Linux']
print('Original List:', os)
os.reverse()
print('Updated List:', os)
```

Output:

```
Original List: ['Windows', 'macOS', 'Linux']
Updated List: ['Linux', 'macOS', 'Windows']
```



PYTHON

LECTURE 28



Today's Agenda



- **List -V**
 - List Comprehension



What Is Comprehension ?



- Comprehensions are **constructs** that allow **sequences** to be built from other sequences.

- In simple words to build a **List** from another **List** or **Set** from another **Set** , we can use **Comprehensions**

- **Python 2.0** introduced **list comprehensions** and **Python 3.0** comes with **dictionary** and **set comprehensions**.

Understanding List Comprehension



- To understand **List comprehensions** let's take a **programming challenge**.
- Suppose you want to take the letters in the word “**Bhopal**”, and put them in a **list**.
- **Can you tell in how many ways can you do this ?**
- Till now , we know **2 ways** to achieve this:
 - **Using for loop**
 - **Using lambda**



Using “for” Loop



```
text="Bhopal"  
myList=[]  
for x in text:  
    myList.append(x)  
print(myList)
```

Output:

```
[ 'B' , 'h' , 'o' , 'p' , 'a' , 'l' ]
```



Using Lambda



```
myList=list(map(lambda x:x , "Bhopal"))
print(myList)
```

Output:

```
[ 'B' , 'h' , 'o' , 'p' , 'a' , 'l' ]
```

Understanding List Comprehension



- Now , we can solve the same problem by using **List Comprehension** also .

- The advantage is that ***List Comprehensions are 35% faster than FOR loop and 45% faster than map function***

Understanding List Comprehension



- To understand this , look at the same code using **List Comprehension**:

```
myList=[x for x in "Bhopal"]  
print(myList)
```

Output:

```
['B', 'h', 'o', 'p', 'a', 'l']
```

Syntax For List Comprehension



□ Syntax:

list_variable = [x for x in iterable <test_cond>]

□ Explanation

- For a Python **List Comprehension**, we use the delimiters for a **list-square brackets**.
- Inside those, we use a **for-statement** on an **iterable**.
- Then there is an **optional test condition** we can apply on each member of **iterable**
- Finally we have our **output expression**



Exercise



- Write a program to produce **square** of nos from **1 to 5** , store them in a **list** and **print** the list.

Do the code using:

1. **Normal for loop**
2. **List Comprehension**

Output:

[1, 4, 9, 16, 25]



Using **for** Loop



```
squaresList=[]
for i in range(1,6):
    squaresList.append(i**2)

print(squaresList)
```



Using List Comprehension



```
squaresList=[x**2 for x in range(1,6)]  
print(squaresList)
```



Exercise



- Write a program to accept a string from the user and **convert** each word of the given string to **uppercase** , store it in a **list** and **print the list**

Do the code using:

1. **Normal for loop**
2. **List Comprehension**

Output:

```
Type a string:my name is sachin
['MY', 'NAME', 'IS', 'SACHIN']
```



Using **for** Loop



```
text=input("Type a string:")
uppersList=[]
for x in text.split():
    uppersList.append(x.upper())
print(uppersList)
```



Using List Comprehension



```
text=input("Type a string:")
uppersList=[x.upper()for x in text.split()]
print(uppersList)
```



Exercise



- Write a program to accept 5 integers from the user and **store** them in a **list** . Now display the **list** and **sum** of the elements.

Do the code using:

1. **Normal for loop**
2. **List Comprehension**

Output:

```
Enter 5 integers:10 20 30 40 50
List is: [10, 20, 30, 40, 50]
Sum is: 150
```



Using **for** Loop

```
myNums=[]
text=input("Enter 5 integers:")
for x in text.split():
    myNums.append(int(x))

print("List is:",myNums)
print("Sum is:",sum(myNums))
```



Using List Comprehension

```
text=input("Enter 5 integers:")  
myNums=[int(x) for x in text.split()]
```

```
print("List is:",myNums)  
print("Sum is:",sum(myNums))
```

Adding Conditions In List Comprehension



- As previously mentioned , it is possible to add a **test condition** in **List Comprehension**.

- When we do this , we get only those items from the **iterable** for which the condition is **True**.

- **Syntax:**

list_variable = [x for x in iterable <test_cond>]



Exercise



- Write a program to produce square of **only odd nos from 1 to 5** , store them in a **list** and **print** the list.

Solution

```
squaresList=[x**2 for x in range(1,6) if x%2!=0]  
print(squaresList)
```

Output:

```
[1, 9, 25]
```



Exercise



- Create a function called **removevowels()** which accepts a **string as argument** and returns a **list** with **all the vowels removed from that string**

Do this code using:

- Normal for loop
- List Comprehension

Output:

```
Type a string:my name is sachin
['m', 'y', ' ', 'n', 'm', ' ', 's', ' ', 's', 'c', 'h', 'n']
```



Using **for** Loop

```
def removevowels(text):
    myList=[]
    for x in text:
        if x not in "aeiou":
            myList.append(x)
    return myList
```

```
text=input("Type a string:")
finalList=removevowels(text)
print(finalList)
```



Using List Comprehension

```
def removevowels(text):
    myList=[x for x in text if x not in "aeiou"]
    return myList
```

```
text=input("Type a string:")
finalList=removevowels(text)
print(finalList)
```



Exercise



- Create a function called **get_numbers()** which accepts a list of **strings , symbols and numbers as argument** and **returns a list containing only numbers from that list.**

Output:

```
Actual List
['bhopal', 25, '$', 'hello', 34, 21, 'indore', 22]
List with numbers only
[25, 34, 21, 22]
```



Solution

```
def get_numbers(myList):
    mynumberList=[x for x in myList if type(x) is int]
    return mynumberList

myList=["bhopal",25,"$","hello",34,21,"indore",22]
print("Actual List")
print(myList)
print("List with numbers only")
mynumberList=get_numbers(myList)
print(mynumberList)
```



Exercise



- Create a function called **getlength()** which accepts a **string** as **argument** and **returns a list** containing the **length** of all the **words** of that string except the word “**the**”. Accept the string from the user.

Output:

```
Type a string:the city of Bhopal is the second cleanest city in the country  
[4, 2, 6, 2, 6, 8, 4, 2, 7]
```



Solution



```
def getlength(str):
    myList=[len(x) for x in str.split() if x!="the"]
    return myList
```

```
text=input("Type a string:")
myList=getlength(text)
print(myList)
```

Adding Multiple Conditions In List Comprehension



- List Comprehension allows us to mention **more than one if condition**.

- To do this we simply have to mention the **next if** after the condition of **first if**

- **Syntax:**

```
list_variable = [x for x in iterable <test_cond_1> <test cond 2>]
```



Exercise



- Write a program to produce square of only those numbers which are divisible by 2 as well as 3 from 1 to 20 , store them in a list and print the list.

Solution

```
myList=[x**2 for x in range(1,21) if x%2==0 if x%3==0]  
print(myList)
```

Output:

```
[36, 144, 324]
```



Previous Code Using **for** Loop

```
myNums=[]
for x in range(1,21):
    if x%2==0:
        if x%3==0:
            myNums.append(x**2)
print(myNums)
```



Exercise



- Write a function called **get_upper()** which accepts a **string** as **argument** and returns a **list** containing only **upper case letters but without any vowels** of that string. Accept the string from the user as input

Output:

```
Type a string:I Live In Bhopal
['L', 'B']
```

```
Type a string:My Name Is Sachin
['M', 'N', 'S']
```



Solution



```
def get_upper (text):  
    myList=[x for x in text if 65<=ord(x)<=90 if x not in "AEIOU"]  
    return myList
```

```
text=input("Type a string:")  
myList=get_upper(text)  
print(myList)
```

What About Logical Operators ?



- **List Comprehension** allows us to use **logical or/and** operator also but we should use **only 1 if statement**

Example:

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

```
b = [x for x in a if x % 2 == 0 or x % 3 == 0]
```

```
print(b)
```

Output:

```
[2, 3, 4, 6, 8, 9, 10]
```



Exercise



- Write a function called **remove_min_max()** which accepts a **list** as **argument** and removes the **minimum** and **maximum** elements from the **list** and returns it

Output:

Original list

```
[10, 3, 15, 12, 24, 6, 1, 18]
```

After removing min and max element

```
[10, 3, 15, 12, 6, 18]
```



Solution



```
def remove_min_max(myList):  
    myNewList=[x for x in myList if x!=min(myList) and x!=max(myList)]  
    return myNewList
```

```
a=[10,3,15,12,24,6,1,18]  
print("Original list")  
print(a)  
print("After removing min and max element")  
print(remove_min_max(a))
```

If – else In List Comprehension



- **List Comprehension** allows us to put **if- else** statements also

- But since in a comprehension, the first thing we specify is the value to put in a list, so we put our **if-else** in place of value.

- **Syntax:**

```
list_variable = [expr1 if cond else expr2  for x in iterable ]
```



Example



```
myList=["Even" if i%2==0 else "Odd" for i in range(1,11)]  
print(myList)
```

Output:

```
['Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even']
```

Nested List Comprehension



- **List Comprehension** can be nested also.
- To understand this , look at the code in the next slide and figure out it's output

Nested List Comprehension



```
a=[20,40,60]
```

```
b=[2,4,6]
```

```
c=[]
```

```
for x in a:
```

```
    for y in b:
```

```
        c.append(x * y)
```

```
print(c)
```

Explanation:

The code is multiplying every element of list **a** , with every element of list **b** and storing the product in list **c**

Output:

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```

Nested List Comprehension



- Now the same code can be rewritten using **Nested List Comprehension**.

- To do this , we will condense each of the lines of code into one line, beginning with the **x * y** operation.

- This will be followed by the **outer for loop**, then the **inner for loop**.



Example



```
a=[20,40,60]
```

```
b=[2,4,6]
```

```
c=[x * y for x in a for y in b]
```

```
print(c)
```

Output:

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```



Exercise



- Write a function called **flatten()** which accepts a nested list as argument and returns a single list containing all the elements of the nested list

Output:

```
Before calling flatten list is
[[1, 2, 3], [4, 5, 6], [7, 8]]
After calling flatten list is
[1, 2, 3, 4, 5, 6, 7, 8]
```



Solution



```
def flatten(mylist):
    newlist=[y for x in mylist for y in x]
    return newlist
```

```
mylist = [[1,2,3],[4,5,6],[7,8]]
print("Before calling flatten list is")
print(mylist)
newlist=flatten(mylist)
print("After calling flatten list is")
print(newlist)
```



PYTHON

LECTURE 29



Today's Agenda



• **Tuple-I**

- What Is A Tuple ?
- Differences With List
- Benefits Of Tuple
- Creating Tuple
- Packing / Unpacking A Tuple
- Accessing A Tuple



What Is A Tuple ?



- Python provides another type that is an **ordered collection of objects**.
- This type is called a **tuple**.



Differences With List



- **Tuples** are identical to **lists** in all respects, except for the following properties:
- **Tuples** are defined by enclosing the elements in **parentheses** `()` instead of **square brackets** `[]`.
- **Tuples** are **immutable**.



Advantages Of Tuple Over List



- **Program execution is faster** when manipulating a **tuple** than it is for the equivalent **list**.
- They prevent **accidental modification**.
- There is another **Python** data type called a **dictionary**, which requires as one of its components a value that is of an **immutable** type. A **tuple** can be used for this purpose, whereas a **list** can't be.



How To Create A Tuple ?



- As mentioned before a tuple is created by placing all the items (elements) inside a parenthesis (), separated by **commas**.
- It can contain **heterogeneous** elements also.

```
# empty tuple
```

```
my_tuple = ()
```

```
# tuple having integers
```

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

```
# tuple with mixed datatypes
```

```
my_tuple = (1, "Hello", 3.4)
```



Guess The Output ?

```
my_tuple = (2)
print(my_tuple)
print(type(my_tuple))
```

Output:

```
2
<class 'int'>
```

Why did it happen ?

Since parentheses are also used to define operator precedence in expressions , so , Python evaluates the expression (2) as simply the integer 2 and creates an int object.

So , How To Create A 1 Element Tuple ?



- To tell **Python** that we really want to define a **singleton tuple**, include a **trailing comma** (,) just before the closing parenthesis.

```
my_tuple = (2,)  
print(my_tuple)  
print(type(my_tuple))
```

Output:

```
(2,)  
<class 'tuple'>
```



Guess The Output ?



```
my_tuple = ()  
print(my_tuple)  
print(type(my_tuple))
```

Output:

```
()  
<class 'tuple'>
```

Packing And Unpacking A Tuple



- **Packing** and **unpacking** is another thing which is commonly used with **tuples**.

Packing:

- **Packing** is a simple syntax which lets us create tuples "on the fly" without using the **normal notation**:
a = 1, 2, 3
- This creates a tuple of 3 values and assigned it to a. Comparing this to the "**normal**" way:
a = (1, 2, 3)

Packing And Unpacking A Tuple



□ Unpacking:

- We can also go the other way, and **unpack** the values from a **tuple** into separate variables:

a = 1, 2, 3

x, y, z = a

- After running this code, we have **x = 1**, **y = 2** and **z = 3**.
- The value of the **tuple a** is unpacked into the 3 variables **x**, **y** and **z**.
- Note that the **number of variables** has to exactly match the **number of elements** in the **tuple**, or there will be an **exception**.

Uses Of Packing And Unpacking



- In the previous code , the variable **a** is just used as a **temporary store** for the **tuple**.

- We also can leave the **middle-man** and do this:
x, y, z = 1, 2, 3

- After running this code, as before, we have **x = 1**, **y =2** and **z = 3**.

- **But can you guess what Python is doing internally ?**

Internal Working



- First the values are packed into a **tuple**:

$(1, 2, 3)$
x, y, z = 1, 2, 3



- Next, the **tuple** is assigned to the left hand side of the = sign:

$\leftarrow (1, 2, 3) \quad (1, 2, 3)$



x, y, z = 1, 2, 3

- Finally, the **tuple** is unpacked into the variables:

$(1, 2, 3)$
x, y, z = 1, 2, 3





Swapping Of 2 Variables



- We have seen the following code many times before which **swaps** the variables **a** and **b** :

a = 10

b = 20

b, a = a, b

- **Now , can you tell what is happening internally ?**

- First, **a** and **b** get packed into a **tuple**.
- Then the **tuple** gets **unpacked** again, but this time into variables **b** then **a**.
- So the values get swapped!

Returning Multiple Values From Function



```
def calculate(a,b):
    c=a+b
    d=a-b
    return c,d

x,y=calculate(5,3)
print("Sum is",x,"and difference is",y)

z=calculate(15,23)
print("Sum is",z[0],"and difference is",z[1])
```

Here **Python** will do the following:

1. It will **pack** the values of **c** and **d** in a **tuple**
2. Then it will **unpack** this **tuple** into the variables **x** and **y**
3. In the **second call** since **z** is a **single variable**, it is automatically converted into a **tuple**

Output:

```
Sum is 8 and difference is 2
Sum is 38 and difference is -8
```



Guess The Output ?

```
def add(a,b,c,d):  
    print("Sum is",a+b+c+d)
```

```
mytuple=(10,20,30,40)  
add(mytuple)
```

Output:

```
add(mytuple)  
TypeError: add() missing 3 required positional arguments: 'b', 'c', and 'd'
```

Why did it happen ?

Since we are passing mytuple as a single argument , so Python is giving exception.

What is the solution ?

To overcome this we must unpack mytuple to 4 individual values. This can be done by prefixing mytuple with a *. So the call will be add(*mytuple)



Solution



```
def add(a,b,c,d):  
    print("Sum is",a+b+c+d)
```

```
mytuple=(10,20,30,40)  
add(*mytuple)
```

Output:

```
Sum is 100
```



Accessing The Tuple



- Similar to a **list** , we can access/print a **tuple** in **three** ways:
 - Directly passing it to the **print()** function
 - Accessing individual elements using **subscript operator []**
 - Accessing multiple elements using **slice operator [:]**



Printing The Entire Tuple



```
values=(10,20,30,40)  
print(values)
```

Output:

```
(10, 20, 30, 40)
```

Accessing Individual Elements



- Like a **list**, a **tuple** also has indexes running from **0** to **size-1**

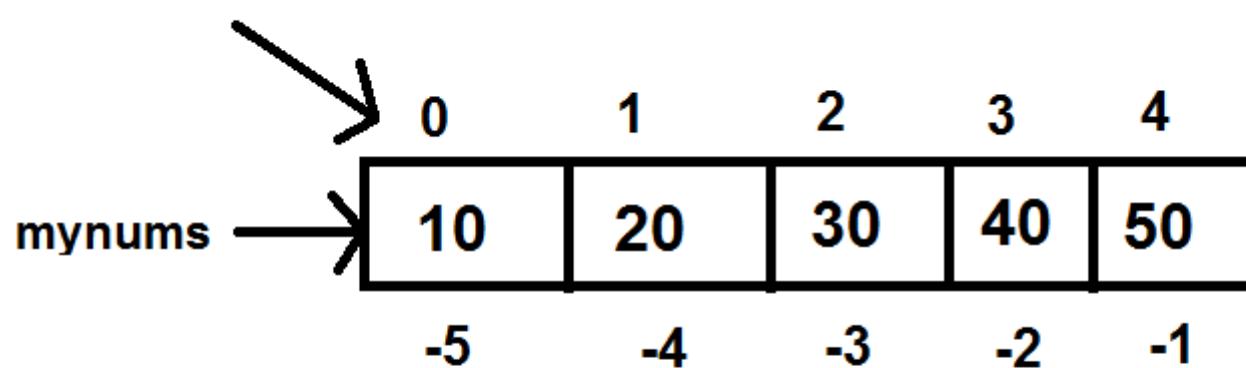
For example:

- `mynums=(10,20,30,40,50)`
- The above code will create a logical diagram in memory, where **positive indexing** will go from **0** to **4** and **negative indexing** from **-1** to **-5**

Accessing Individual Elements



Forward Indexing



Backward Indexing

Accessing Individual Elements



```
mynums=(10,20,30,40,50)  
print(mynums[0])  
print(mynums[1])  
print(mynums[-3])  
print(mynums[-2])
```

Output:

```
10  
20  
30  
40
```

Even though we create tuple using the symbol of () but when we access it's individual element , we still use the subscript or index operator []

Accessing Tuple Elements Using While Loop



```
mynums=(10,20,30,40,50)
```

```
n=len(mynums)
```

```
i=0
```

```
while i<n:
```

```
    print(mynums[i])
```

```
    i=i+1
```

Output:

```
10  
20  
30  
40  
50
```

Just like **len()** works with **list**, similarly it also works with **tuple** and returns **number of elements in the tuple**

Accessing Tuple Elements Using For Loop



```
mynums=(10,20,30,40,50)
```

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

Output:

```
10  
20  
30  
40  
50
```

Since **tuple** is a sequence type , so for loop can iterate over individual elements of the tuple



Exercise

Given the tuple below that represents the **Arijit Singh's Aashiqui 2 songs**.

Write code to print the album details, followed by a listing of all the tracks in the album.

```
album="Aashiqui 2", 2013 , "Arijit Singh",((1,"Tum hi ho"),(2,"Chahun Mai Ya Na"),(3,"Meri Aashiqui"),(4,"Aasan Nahin Yahaan"))
```

Output:

Title: Aashiqui 2

Year: 2013

Singer: Arijit Singh

 Song Number:1,Song Name:Tum hi ho

 Song Number:2,Song Name:Chahun Mai Ya Na

 Song Number:3,Song Name:Meri Aashiqui

 Song Number:4,Song Name:Aasan Nahin Yahaan



Solution

```
album="Aashiqui 2","Arijit Singh",2013,((1,"Tum hi  
ho"),(2,"Chahun Mai Ya Na"),(3,"Meri  
Aashiqui"),(4,"Aasan Nahin Yahaan"))
```

```
title,singer,year,songs=album  
print("Title:",title)  
print("Year:",year)  
print("Singer:",singer)  
for info in songs:  
    print("\tSong Number:{0},Song  
Name:{1}".format(info[0],info[1]))
```



Slice Operator With Tuple



- Just like we can apply slice operator with **lists** and **strings** , similarly **Python** allows us to apply slice operator with **tuple** also.

- **Syntax:** `tuple_var[x:y]`
 - **x** denotes the **start index** of slicing and **y** denotes the **end index** . But **Python** ends slicing at **y-1** index.



The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[1:4])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[3:5])
```

- **Output:**

(20,30,40)

- **Output:**

(40,50)



The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[0:4])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[0:10])
```

- **Output:**

(10,20,30,40)

- **Output:**

(10,20,3040,50)



The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[2:2])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[6:10])
```

- **Output:**

()

- **Output:**

()



The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[1: ])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[:3])
```

- **Output:**

(20,30,40,50)

- **Output:**

(10,20,30)



The Slicing Operator

- **Example:**

```
Mynums=(10,20,30,40,50)  
print(mynums[ :-2])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[-2:])
```

- **Output:**

(10, 20,30)

- **Output:**

(40,50)



Using Step Value

- The concept of **step value** in slicing with **tuple** is also same as that with **list**
 - Movement is in forward direction (L \square R)
 - Default for start is **0** and end is **len**
- But if **step** is **negative** , then
 - Movement is in backward direction (R \square L)
 - Default for start is **-1** and end is **-(len+1)**



The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[1:4:2])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[1:4:0])
```

- **Output:**
(20,40)

- **Output:**
**ValueError: Slice
step cannot be
0**



The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[4:1:-1])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[4:1:-1])
```

- **Output:**

()

- **Output:**

(50,40,30)



The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[::-1])
```

- **Example:**

```
Mynums=(10,20,30,  
        40,50)  
print(mynums[::-1])
```

- **Output:**

(10,20,30,40,50)

- **Output:**

(50,40,30,20,10)



PYTHON

LECTURE 30



Today's Agenda



- **Tuple-II**
 - Changing The Tuple
 - Deleting The Tuple
 - Functions Used With Tuple



Changing A Tuple



- Unlike **lists**, **tuples** are **immutable**.
- This means that elements of a **tuple** *cannot be changed* once it has been assigned.



Guess The Output ?



`mynums=(10,20,30,40,50)`

`mynums[0]=15`

Output:

```
mynums[0]=15
TypeError: 'tuple' object does not support item assignment
```



Guess The Output ?



```
mynums=[10,20],30,40,50)
```

```
print(mynums)
```

```
mynums[0][0]=15
```

```
print(mynums)
```

Why did the code run?

**Although a tuple is immutable,
but if it contains a mutable
data then we can change
it's value**

Output:

```
([10, 20], 30, 40, 50)
([15, 20], 30, 40, 50)
```



Guess The Output ?



```
myvalues=("hello",30,40,50)
print(myvalues)
myvalues[0]="hi"
print(myvalues)
```

Output:

```
('hello', 30, 40, 50)
Traceback (most recent call last):
  File "tupledemo12.py", line 3, in <module>
    myvalues[0]="hi"
TypeError: 'tuple' object does not support item assignment
```



Guess The Output ?



```
myvalues=[["hello",20],30,40,50)  
print(myvalues)  
myvalues[0][0]="hi"  
print(myvalues)
```

Output:

```
(['hello', 20], 30, 40, 50)  
(['hi', 20], 30, 40, 50)
```



Guess The Output ?



```
mynums=(10,20,30,40,50)
```

```
print(mynums)
```

```
mynums=(15,25,35,45,55)
```

```
print(mynums)
```

Why did the code run?

Tuple object **is immutable** ,
but tuple reference **is mutable**.
**So we can assign a new
tuple object to the same reference**

Output:

```
(10, 20, 30, 40, 50)
(15, 25, 35, 45, 55)
```



Deleting A Tuple



- As we discussed previously, a **tuple** is **immutable**.
- This also means that we can't **delete** just a part of it.
- However we can **delete** an **entire tuple** if required.



Guess The Output ?



```
mynums=(10,20,30,40,50)
print(mynums)
del mynums[0]
print(mynums)
```

Output:

```
(10, 20, 30, 40, 50)
Traceback (most recent call last):
  File "tupledemo14.py", line 3, in <module>
    del mynums[0]
TypeError: 'tuple' object doesn't support item deletion
```



Guess The Output ?



```
mynums=(10,20,30,40,50)
print(mynums)
del mynums[2:4]
print(mynums)
```

Output:

```
Traceback (most recent call last):
  File "tupledemo14.py", line 3, in <module>
    del mynums[2:4]
TypeError: 'tuple' object does not support item deletion
```



Guess The Output ?



```
mynums=(10,20,30,40,50)
print(mynums)
del mynums
print(mynums)
```

Output:

```
(10, 20, 30, 40, 50)
Traceback (most recent call last):
  File "tupledemo14.py", line 4, in <module>
    print(mynums)
NameError: name 'mynums' is not defined
```



Built In Functions For Tuple



- A lot of functions that work on **lists** work on **tuples** too.
- But only those functions work with **tuple** which do not **modify** it
- **Can you figure out which of these functions will work with tuple ?**

Answer:

- `len()`
- `max()`
- `min()`
- `sum()`
- `sorted()`
- `tuple()`
- `any()`
- `all()`

All of them will work with tuple.

Will `sorted()` also work ?

Yes, even `sorted()` function will also work since it does not change the original tuple , rather it returns a sorted copy of it



The **len()** Function

- Returns the **number of items** in the **tuple**

Example:

```
fruits=("apple","banana","orange",None)  
print(len(fruits))
```

Output:

4



The **max()** Function



- Returns the **greatest** item present in the **tuple**

Example:

```
nums=(5,2,11,3)
```

```
print(max(nums))
```

Output:

11



Guess The Output ?



```
months=("january","may","december")
print(max(months))
```

Output:

may



Guess The Output ?



```
booleans=(False,True)  
print(max(booleans))
```

Output:

True



Guess The Output ?



Mynums=(True,5,False)
print(max(mynums))

Output:

5



Guess The Output ?



```
mynums=("True",False)  
print(max(mynums))
```

Output:

```
print(max(mynums))  
TypeError: '>' not supported between instances of 'bool' and 'str'
```



Guess The Output ?



```
values=(10,"hello",20,"bye")
print(max(values))
```

Output:

```
print(max(values))
TypeError: '>' not supported between instances of 'str' and 'int'
```



Guess The Output ?



```
fruits=("apple","banana","orange")
print(max(fruits))
```

Output:

orange



Guess The Output ?



```
fruits=("apple","banana","orange",None)  
print(max(fruits))
```

Output:

```
print(max(fruits))  
TypeError: '>' not supported between instances of 'NoneType' and 'str'
```



The **min()** Function



- Returns the **least** item present in the **tuple**

Example:

```
nums=(5,2,11,3)
```

```
print(min(nums))
```

Output:

2



Guess The Output ?



```
months=("january","may","december")
print(min(months))
```

Output:

december



The **sum()** Function



- Returns the **sum** of all the **items** present in the **tuple** .
- As before , the items must be of **Numeric** or **boolean** type

Example:

```
nums=(10,20,30)  
print(sum(nums))
```

Output:

60



Guess The Output ?



```
nums=(10,20,30,True,False)  
print(sum(nums))
```

Output:

61



Guess The Output ?



```
nums=('1','2','3')  
print(sum(nums))
```

Output:

```
print(sum(nums))  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



Guess The Output ?



```
nums=(2.5,3.5,4.5)  
print(sum(nums))
```

Output:

10.5



The **sorted()** Function



- Returns a **sorted version** of the **tuple** passed as argument.

Example:

```
nums=(7,4,9,1)  
print(sorted(nums))  
print(nums)
```

Did you notice a special point ?

Although **sorted()** is working on a **tuple** , but it has returned a **list**

Output:

```
[1, 4, 7, 9]  
(7, 4, 9, 1)
```



Guess The Output ?



```
months=("january","may","december")
print(sorted(months))
```

Output:

[“december”, “january”, “may”]



Guess The Output ?



```
months=("january","may","december",3)  
print(sorted(months))
```

Output:

```
print(sorted(months))  
TypeError: '<' not supported between instances of 'int' and 'str'
```



Guess The Output ?



```
values=(2.4,1.0,2,3.6)  
print(sorted(values))
```

Output:

[1.0,2,2.4,3.6]



Guess The Output ?



```
values=("bhupal","bhop","Bhopal")
print(sorted(values))
```

Output:

["Bhopal", "bhop", "bhupal"]



Sorting In Descending Order

- To **sort** the **tuple** in **descending order** , we can pass the **keyword argument reverse** with value set to **True** to the function **sorted()**

Example:

```
nums=(7,4,9,1)  
print(sorted(nums,reverse=True))  
print(nums)
```

Output:

```
[9, 7, 4, 1]  
(7, 4, 9, 1)
```



The **tuple()** Function



- The **tuple()** function converts an **iterable** i.e **list** , **range**, **set** , **dictionary** and **string** to a **tuple**.

Syntax:

tuple(iterable)

Example:

```
city="bhopal"  
x=tuple(city)  
print(x)
```

Output:

```
('b', 'h', 'o', 'p', 'a', 'l')
```



Guess The Output ?



n=20

x=tuple(n)

print(x)

Output:

```
x=tuple(n)
```

```
TypeError: 'int' object is not iterable
```



Guess The Output ?



```
n="20"  
x=tuple(n)  
print(x)
```

Output:

```
('2', '0')
```



Guess The Output ?



```
l=[10,20,30]  
x=tuple(l)  
print(x)
```

Output:

```
(10, 20, 30)
```



The **any()** Function

- The **any()** function accepts a **Tuple** as argument and returns **True** if atleast **one element** of the **Tuple** is **True**. If not, this method returns **False**. If the **Tuple** is empty, then also it returns **False**

Syntax:

list(iterable)

Example:

```
x = (1, 3, 4, 0)  
print(any(x))
```

Output:

True



Guess The Output ?



```
x = (0, False)  
print(any(x))
```

Output:

False



Guess The Output ?



```
x = (0, False, 5)  
print(any(x))
```

Output:

True



Guess The Output ?



x= ()

print(any(x))

Output:

False



Guess The Output ?



```
x=("", "o", "")  
print(any(x))
```

Output:

True



Guess The Output ?



```
x=("",0, "")  
print(any(x))
```

Output:

False



Guess The Output ?



```
x= (4)  
print(any(x))
```

Output:

```
print(any(x))  
TypeError: 'int' object is not iterable
```



Guess The Output ?



```
x= (4,)  
print(any(x))
```

Output:

True



The **all()** Function



- The **all()** function accepts a **Tuple** as argument and returns **True** if **all the elements** of the **Tuple** are **True** or if the **Tuple** is **empty**. If not, this method returns **False**.

Syntax:

all(iterable)

Example:

```
x = (1, 3, 4, 0)  
print(all(x))
```

False



Guess The Output ?



```
x = (0, False)  
print(all(x))
```

Output:

False



Guess The Output ?



```
x = (1,3,4,5)  
print(all(x))
```

Output:

True



Guess The Output ?



```
x = (0, False, 5)  
print(all(x))
```

Output:

False



Guess The Output ?



x= 0

print(all(x))

Output:

True



These Notes Have Python_world_InNotes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 31



Today's Agenda

- **Tuple-III**
 - Methods Used With Tuple
 - Operations Allowed On Tuple



Tuple Methods

- As mentioned previously a **Tuple** is **immutable**.
- So only those methods work with it which *do not change* the **tuple** data
- **Can you figure out which of these methods work with tuples ?**
- These are:
 - **append()**
 - **extend()**
 - **insert()**
 - **index()**
 - **count()**
 - **remove()**
 - **pop()**
 - **clear()**
 - **sort()**
 - **reverse()**

Answer:

Only index() and count().
Rest all the methods change the sequence object on which they have been called.



The **index()** Method

- The **index()** method searches an element in the **tuple** and returns it's **index**.
- If the element occurs **more than once** it returns it's **smallest/first position**.
- If element is **not found**, it raises a **ValueError** exception

Syntax:

`tuple_var.index(item)`

Example:

`primes=(2,3,5,7)`

`pos=primes.index(5)`

`print("position of 5 is",pos)`

Output:

position of 5 is 2



Guess The Output ?

```
mynums = (10,20,30,40,50)
```

```
p = mynums.index(20)
```

```
print("20 occurs at",p,"position")
```

```
p = mynums.index(60)
```

```
print("60 occurs at",p,"position")
```

```
p = mynums.index(10)
```

```
print("10 occurs at",p,"position")
```

Output:

```
20 occurs at 1 position
Traceback (most recent call last):
  File "tupledemo15.py", line 4, in <module>
    p = mynums.index(60)
ValueError: tuple.index(x): x not in tuple
```



The **count()** Method



- The **count()** method returns the **number of occurrences** of an element in a **tuple**
- In simple terms, it **counts** how many times an element has occurred in a **tuple** and returns it.

Syntax:

tuple_var.count(item)

Example:

```
country=('i','n','d','i','a')
```

```
x=country.count('i')
```

```
print("i occurs",x,"times in",country)
```

Output:

```
i occurs 2 times in ('i', 'n', 'd', 'i', 'a')
```



Guess The Output ?

```
vowels = ('a', 'e', 'i', 'o', 'i', 'u')
x = vowels.count('i')
print("i occurs",x,"times")
x = vowels.count('e')
print("e occurs",x,"times")
x = vowels.count('j')
print("j occurs",x,"times")
```

Output:

```
i occurs 2 times
e occurs 1 times
j occurs 0 times
```



Operations Allowed On Tuple



- We can apply **four** types of operators on **Tuple** objects
- These are:
 - **Membership Operators**
 - **Concatenation Operator**
 - **Multiplication**
 - **Relational Operators**



Membership Operators

- We can apply the '**in**' and '**not in**' operators on **tuple**.
- This tells us whether an item **belongs / not belongs** to **tuple**.



Guess The Output ?

```
my_tuple = ('a','p','p','l','e',)  
print('a' in my_tuple)  
print('b' in my_tuple)  
print('g' not in my_tuple)
```

Output:

True
False
True



Concatenation On Tuple



- **Concatenation** is the act of joining.
- We can join two **tuples** using the **concatenation operator** ‘+’.
- All other arithmetic operators are not allowed to work on two tuples.
- However * works but as a **repetition operator**



Guess The Output ?



```
odds=(1,3,5)  
evens=(2,4,6)  
all=odds+evens  
print(all)
```

Output:

```
(1, 3, 5, 2, 4, 6)
```



Guess The Output ?

```
ages=(10,20,30)
names=("amit","deepak","ravi")
students=ages+names
print(students)
```

Output:

```
(10, 20, 30, 'amit', 'deepak', 'ravi')
```



Multiplication On Tuple



- Python allows us to **multiply** a **tuple** by a **constant**
- To do this , as usual we use the operator *



Guess The Output ?



a=(10,20,30)

b=a*3

print(b)

Output:

```
(10, 20, 30, 10, 20, 30, 10, 20, 30)
```



Guess The Output ?



a=(10,20,30)

b=a*3.0

print(b)

Output:

```
b=a*3.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Relational Operators On Tuples



- The **relational operators** work with **tuples** and other sequences.
- **Python** starts by comparing the **first element** from each sequence.
- If they are **equal**, it goes on to the **next element**, and so on, until it finds elements that **differ**.
- **Subsequent elements** are not considered (even if they are really big).



Guess The Output ?



a=(1,2,3)

b=(1,3,4)

print(a<b)

Output:

True



Guess The Output ?



```
a=(1,3,2)  
b=(1,2,3)  
print(a<b)
```

Output:

False



Guess The Output ?



```
a=(1,3,2)  
b=(1,3,2)  
print(a<b)
```

Output:

False



Guess The Output ?



```
a=(1,2,3)  
b=(1,2,3,4)  
print(a<b)
```

Output:

True



Guess The Output ?



```
a=(5,2,7)  
b=(1,12,14)  
print(a>b)
```

Output:

True



Guess The Output ?



a=()

b=(0)

print(a<b)

Output:

**TypeError : < not supported between instances of
'tuple' and 'int'**



Guess The Output ?



```
a=0  
b=(0,)  
print(a<b)
```

Output:

True



Guess The Output ?



```
a=(1,2)  
b=("one","two")  
print(a<b)
```

Output:

```
print(a<b)
TypeError: '<' not supported between instances of 'int' and 'str'
```



Guess The Output ?

```
a=(1,"one")
b=(1,"two")
print(a<b)
```

Output:

True



These Notes Have Python_world™ Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 32



Today's Agenda



• **Strings -I**

- What Is A String ?
- Creating A String
- Different Ways Of Accessing Strings
- Operators Which Work On Strings



What Is A String ?



- A Python **string** is a sequence of **zero or more** characters.
- It is an **immutable** data structure.
- This means that we although we **can access** the internal data elements of a string object but we **can not change** it's contents



How Can Strings Be Created ?

- Python provides us 3 ways to create string objects:
 - By enclosing text in **single quotes**
 - By enclosing text in **double quotes**
 - By enclosing text in **triple quotes (generally used for multiline strings)**



Example

```
my_string = 'Hello'  
print(my_string)  
my_string = "Hello"  
print(my_string)  
my_string = ""Hello""  
print(my_string)  
my_string = """Hello, welcome to  
the world of Python"""  
print(my_string)
```

Output:

```
Hello  
Hello  
Hello  
Hello, welcome to  
the world of Python
```

How Can Strings Be Accessed ?



- **Python** provides us **3 ways** to access **string** objects:
 - Directly passing it to the **print()** function
 - Accessing individual elements using **subscript operator []**
 - Accessing multiple elements using **slice operator [:]**



Printing The Whole String



```
city="Bhopal"  
print(city)
```

Output:

Bhopal

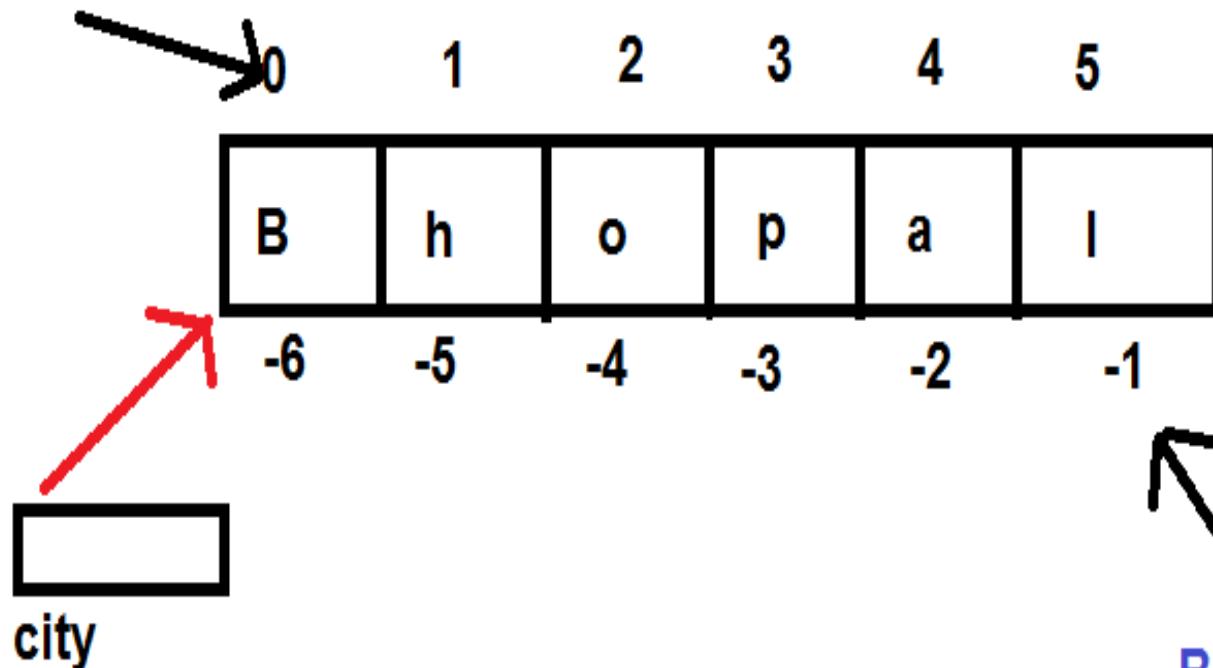
Accessing Individual Elements



- A **string** in **Python** has indexes running from **0** to **size-1**
- **For example:**
 - **city=“Bhopal”**
- The above code will create a logical diagram in memory, where positive indexing will go from **0** to **5** and negative indexing from **-1** to **-6**

Accessing Individual Element

Forward Indexing



Backward Indexing



Accessing Individual Element

```
city="Bhopal"  
print(city[0])  
print(city[1])  
print(city[-1])  
print(city[-2])
```

Output:

B
h
l
a



Guess The Output ?



```
city="Bhopal"  
print(city[6])
```

Output:

IndexError: String index out of range



Guess The Output ?



```
city="Bhopal"  
print(city[1.5])
```

Output:

TypeError: String indices must be integers

Accessing String Elements Using while Loop



```
city="Bhopal"
```

```
i=0
```

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

```
    print(city[i])
```

```
    i=i+1
```

Output:

```
B  
h  
o  
p  
a  
l
```

Just like **len()** works with **lists** and **tuple** similarly it also works with **string** returns **number of elements** in the **string**

Accessing String Elements Using for Loop



```
city="Bhopal"  
for ch in city:  
    print(ch)
```

Output:

B
h
o
p
a
l

Since **string** is a sequence type , so for loop can iterate over individual elements of the **string**



Exercise



- Redesign the previous code using for loop only to traverse the **string** in reverse order. Don't use slice operator



Solution



```
city="Bhopal"  
for i in range(len(city)-1,-1,-1):  
    print(city[i])
```

Output:

l
a
p
o
h
B



Slice Operator With String

- Just like we can apply slice operator with **lists** and **tuples**, similarly **Python** allows us to apply slice operator with **strings** also.
- Syntax:** `string_var[x:y]`
 - x** denotes the **start index** of slicing and **y** denotes the **end index**. But **Python** ends slicing at **y-1** index.



The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[1:4])
```

- **Example:**

```
city="Bhopal"  
print(city[3:5])
```

- **Output:**

hop

- **Output:**

pa



The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[0:4])
```

- **Example:**

```
city="Bhopal"  
print(city[0:10])
```

- **Output:**

Bhop

- **Output:**

Bhopal



The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[1:])
```

- **Example:**

```
city="Bhopal"  
print(city[:4])
```

- **Output:**

hopal

- **Output:**

Bhop



The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[:-2])
```

- **Example:**

```
city="Bhopal"  
print(city[-2:])
```

- **Output:**

Bhop

- **Output:**

al



Using Step Value

- String slicing can accept a **third parameter** also after the two index numbers.
- The **third parameter** is called **step value**.
- So the complete syntax of slicing operator is:

s[begin:end:step]

- Step value indicates *how many characters to move forward after the first character is retrieved* from the string and it's default value is **1** , but can be changed as per our choice.



Using Step Value

- Another point to understand is that if **step** is **positive** or **not mentioned** then
 - **Movement is in forward direction (L→R)**
 - **Default for start is 0 and end is len**
- But if **step** is **negative** , then
 - **Movement is in backward direction (R→L)**
 - **Default for start is -1 and end is -(len+1)**



The Slicing Operator

- Example:

```
city="Bhopal"  
print(city[1:4:2])
```
- Output:
`hp`
- Example:

```
city="Bhopal"  
print(city[1:4:0])
```
- Output:
`ValueError: Slice
step cannot be
0`



The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[4:1:-1])
```

- **Example:**

```
city="Bhopal"  
print(city[4:1:-1])
```

- **Output:**

apo



The Slicing Operator

- Example:

```
city="Bhopal"  
print(city[::-1])
```
- Output:
Bhopal
- Example:

```
city="Bhopal"  
print(city[::-1])
```
- Output:
lapohB

The Operators With Strings



- There are **6** operators which work on **Strings**:
 - **+** : For joining 2 strings
 - ***** : For creating multiple copies of a string
 - **in** : For searching a substring in a string
 - **not in**: Opposite of in
 - **Relational Operator** : For comparing 2 strings
 - **Identity Operators** : For comparing addresses



The Operator +

The **+** operator concatenates strings. It returns a **string** consisting of the operands joined together

Example:

a="Good"

b="Morning"

c="User"

print(a+b+c)

Output:

GoodMorningUser



The Operator *

- The * operator creates multiple copies of a **string**.

Example:

```
a="Bye"  
print(a*2)  
print(2*a)
```

Output:

```
ByeBye  
ByeBye
```



Guess The Output ?

```
a="Bye"  
print(a*0)  
print(a*-2)
```

Output:

The ***** operator
allows its
operand to be
negative or 0 in
which case it
returns an **empty**
string



Guess The Output ?



```
x="Ba"+"na"*2  
print(x)
```

Output:
Banana



The Operator in



- The **in** operator returns **True** if the first operand is contained within the second, and **False** otherwise:

Example:

a="banana"

print("nana" in a)

print("nani" in a)

Output:

True

False



The Operator not in

- The **not in** operator behaves opposite of **in** and returns **True** if the first operand is not contained within the second, and **False** otherwise:

Example:

```
a="banana"
```

```
print("nana" not in a)
```

```
print("nani" not in a)
```

Output:

False

True

The Relational Operators



- We can use (`>` , `<` , `<=` , `>=` , `==` , `!=`) to compare two strings.
 - **Python** compares string lexicographically i.e using Unicode value of the characters.

Example:

Output:

"tim" == "tie"

False

"free" \neq "freedom"

True

"arrow" > "aron"

True

"right" >= "left"

True

"teeth" < "tee"

False

"yellow" <= "fellow"

True

"abc" > " "



The Identity Operators



- ‘**is**’ operator returns **True** if both the operand point to same memory location.
- ‘**is not**’ operator returns **True** if both the operand point to different memory location.

Example:

```
a = 'London'  
b = 'London'  
c = 'Paris'  
print(a is b)  
print(a is c)  
print(b is c)  
print(b is not a)  
print(b is not c)
```

Output:

True
False
False
False
True



PYTHON

LECTURE 33

These Notes Have Python _world _In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Today's Agenda



- **Strings -II**

- Built In String Functions
- Printing string using f-string
- Modifying Strings



Built In String Functions



- There are some **built-in functions** in **Python** that we can use on **strings**.
- These are:
 - **len()**
 - **max()**
 - **min()**
 - **chr()**
 - **ord()**



The **len()** Function

- Returns the **number of characters** in the **string**

Example:

```
name="Sachin"  
print(len(name))
```

Output:

6



The **max()** Function

- Returns a character which is **alphabetically the highest character** in the **string**.

Example:

```
name="bhopal"
```

```
print(max(name))
```

Output:

```
p
```



Guess The Output ?



```
str="abc123#$.y@*"  
print(max(str))
```

Output:

y



Guess The Output ?



```
str="False,True"  
print(max(str))
```

Output:

u



Guess The Output ?



```
str="1.1,0.4,1.9"  
print(max(str))
```

Output:

9



The **min()** Function

- Returns a character which is **alphabetically the lowest character** in the **string**.

Example:

```
name="bhopal"
```

```
print(min(name))
```

Output:

```
a
```



Guess The Output ?



str="Bhopal"

print(min(str))

Output:

B



Guess The Output ?



```
str="abc123#$.y@*"  
print(min(str))
```

Output:

#



Guess The Output ?



```
str="1.1,0.4,1.9"  
print(min(str))
```

Output:

,



The **chr()** Function



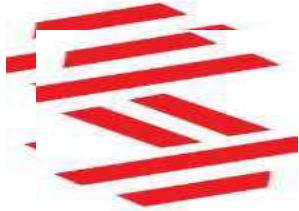
- Returns a **character** (a string) of the **unicode** value passed as an **integer**.
- The valid range of the argument is from **0** through **1,114,111**.

Example:

```
print(chr(122))
```

Output:

z



Guess The Output ?



print(chr(43))

Output:

+



Guess The Output ?



print(chr(1))

Output:





Guess The Output ?



print(chr(0))

Output:



Guess The Output ?

`print(chr(-1))`

Output:

`ValueError: chr() argument not in range`



The `ord()` Function



- Returns an **integer** of the **unicode** value passed as an **character**.
- But the argument passed should be only 1 character in length.

Example:

```
print(ord('a'))
```

Output:

97



Guess The Output ?



print(ord("+"))

Output:

43



Guess The Output ?



print(ord("5"))

Output:

53



The **str()** Function

- Returns a **string** representation of an **object**.
- We can pass object of any type and Python will convert it to string

Example:

```
print(str(49.2))
```

Output:

49.2



Guess The Output ?



print(str(True))

Output:

True



Guess The Output ?



print(str(25))

Output:

25



String Interpolation



- In **Python** version **3.6**, a new string formatting mechanism was introduced.
- This feature is called **String Interpolation** , but is more usually referred to by its nickname **f-string**.



String Interpolation



- To understand this feature , can you tell how can we print the following **2 variables** using **print()**:

name=“Sachin”

age=34

- Till now , we know **4 ways**:

- 1. print(“My name is”,name,”and my age is”,age)**
- 2. print(“My name is ”+name+” and my age is ”+str(age))**
- 3. print(“My name is {0} and my age is {1}”.format(name,age))**
- 4. print(“My name is %s and my age is %d”%(name,age))**



String Interpolation



- But , from **Python 3.6** onwards , there is much more simpler way to print them which is called **f-string**.
- **f-strings** have an **f** at the beginning and **curly braces containing expressions** that will be **replaced** with their values.
- The expressions are evaluated at runtime and then formatted

`name="Sachin"`

`age=34`

`print(f"My name is {name} and my age is {age}")`



Arbitrary Expressions



- Because f-strings are evaluated at runtime, we can put any valid **Python expressions** in them

a=10

b=20

print(f"sum is {a+b}")

Output:

sum is 30



Function Calls



- We could also call functions. Here's an example:

```
import math
```

```
a=10
```

```
b=20
```

```
print(f"max of {a} and {b} is {max(a,b)}")
```

```
print(f"Factorial of {a} is {math.factorial(10)}")
```

Output:

```
max of 10 and 20 is 20
Factorial of 10 is 3628800
```



Method Calls



- We could also call **methods**. Here's an example:

```
vowels=["a","e","i","o","u","a"]
```

```
ch="a"
```

```
print(f"{ch} is occurring in {vowels}  
{vowels.count(ch)} times")
```

Output:

```
a is occurring in ['a', 'e', 'i', 'o', 'u', 'a'] 2 times
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 34



Today's Agenda



- **Strings -III**
 - String Methods



String Methods



- A string object has a number of **method** or **member functions**.
- These can be grouped into different categories .
- These categories are:
 - **String conversion methods**
 - **String comparison methods**
 - **String searching methods**
 - **String replace methods**



String Conversion Methods



- **capitalize()**

Returns a copy of the string with **first character capitalized** and rest of the characters in **lower case**.

Example:

```
name="guido van rossum"  
newname=name.capitalize()  
print(f'Original name is {name}\nCapitalized name is {newname}')
```

Output:

```
Original name is guido van rossum  
Capitalized name is Guido van rossum
```



String Conversion Methods



```
text="python is awesome. java rocks"  
newtext=text.capitalize()  
print(f'Original text is {text}\nCapitalized text is {newtext}')
```

Output:

```
Original text is python is awesome. java rocks  
Capitalized text is Python is awesome. java rocks
```



String Conversion Methods

- **lower()** and **upper()**

Returns a copy of the string with all letters converted to **lowercase** and **uppercase** respectively

Example:

```
name="Sachin Kapoor"  
lc=name.lower()  
uc=name.upper() print(f'Original  
name is {name}') print(f'Lower  
name is {lc}') print(f'Upper name  
is {uc}')
```

Output:

```
Original name is Sachin Kapoor  
Lower name is sachin kapoor  
Upper name is SACHIN KAPOOR
```



String Conversion Methods



- **swapcase()**

Returns a copy of the string with the **case** of every character **swapped**. Means that **lowercase characters** are changed to **uppercase** and **vice-versa**.

Example:

```
name="Sachin Kapoor"  
newname=name.swapcase()  
print(f'Original name is {name}')  
print(f'Swapped name is {newname}')
```

Output:

```
Original name is Sachin Kapoor  
Swapped name is SACHIN kAPOOR
```



String Conversion Methods



- **title()**

Returns a copy of the string converted to **proper case** or **title case**. i.e., all **words** begin with **uppercase letter** and the **rest** are in **lowercase**.

Example:

```
text="we got independence in 1947"
newtext=text.title() print(f"Original
text is {text}") print(f"Title text is
{newtext}")
```

Output:

```
Original text is we got independence in 1947
Title text is We Got Independence In 1947
```



String Conversion Methods

```
text = "i LOVE pYTHON"  
print(text.title())
```

Output:

```
I Love Python
```



String Conversion Methods

```
text = "physics,chemistry,maths"  
print(text.title())
```

Output:

```
Physics,chemistry,Maths
```



String Conversion Methods

```
text = "physics_chemistry_maths"  
print(text.title())
```

Output:

```
Physics_Chemistry_Maths
```



String Conversion Methods

```
text = "physics1chemistry2maths"  
print(text.title())
```

Output:

```
Physics1Chemistry2Maths
```



String Conversion Methods

```
text = "He's an engineer, isn't he?"  
print(text.title())
```

Output:

```
He'S An Engineer, Isn'T He?
```



String Comparison Methods

- **islower() and isupper()**

Returns **True** or **False** depending on whether all alphabets in the string are in **lowercase** and **uppercase** respectively



String Comparison Methods



Example:

```
s = 'this is good'
```

```
print(s.islower())
```

```
s = 'th!s is a1so g00d'
```

```
print(s.islower())
```

```
s = 'this is Not good'
```

```
print(s.islower())
```

Output:

```
True
```

```
True
```

```
False
```



String Comparison Methods



Example:

```
s = "THIS IS GOOD!"
```

```
print(s.isupper())
```

```
s = "THIS IS ALSO GooD!"
```

```
print(s.isupper())
```

```
s= "THIS IS not GOOD!"
```

```
print(s.isupper())
```

Output:

```
True  
True  
False
```



String Comparison Methods



Example:

```
s = ""
```

```
print(s.isupper())
print(s.islower())
```

Output:

```
False
False
```



String Comparison Methods



- **istitle()**

Returns **True** if the string is in **titlecase** or **empty** ,
otherwise returns **False**



String Comparison Methods

Example:

```
s = 'Python Is Good.'  
print(s.istitle())
```

```
s = 'Python is good'  
print(s.istitle())
```

```
s = 'This Is @ Symbol.'  
print(s.istitle())
```

```
s = '99 Is A Number'  
print(s.istitle())
```

```
s = 'PYTHON'  
print(s.istitle())
```

Output:

True
False
True
True
False



String Comparison Methods



- **isalpha()**

Returns **True** if the string contains only **alphabets**,
otherwise returns **False**



String Comparison Methods

Example:

```
name = "Monalisa"  
print(name.isalpha())
```

```
name = "M0nalisa"  
print(name.isalpha())
```

```
name = "Monalisa Shah"  
print(name.isalpha())
```

Output:

True
False
False



String Comparison Methods



- **isdigit()**

Returns **True** if the string contains only **digits** , otherwise returns **False**



String Comparison Methods

Example:

```
text = "12345"
```

```
print(text.isdigit())
```

```
text = "012345"
```

```
print(text.isdigit())
```

```
text = "12345 6"
```

```
print(text.isdigit())
```

```
text = "a12345"
```

```
print(text.isdigit())
```

Output:

True

True

False

False



String Comparison Methods



- **isalnum()**

Returns **True** if the string contains only **alphanumeric** characters , otherwise returns **False**



String Comparison Methods

Example:

```
name = "M234onalisa"  
print(name.isalnum())
```

```
name = "M3ona Shah "  
print(name.isalnum())
```

```
name = "Mo3nalisaSha22ah"  
print(name.isalnum())
```

```
name = "133"  
print(name.isalnum())
```

Output:

True
False
True
True



String Comparison Methods



- **isspace()**

Returns **True** if the string contains only **whitespace** characters , otherwise returns **False**



String Comparison Methods

Example:

```
s = ' \t'
```

```
print(s.isspace())
```

```
s = ' a '
```

```
print(s.isspace())
```

```
s = ''
```

```
print(s.isspace())
```

```
s = "
```

```
print(s.isspace())
```

Output:

True

False

True

False



String Comparison Methods



- **startswith()**
- The **startswith()** method takes maximum of **three** parameters:
 - **prefix** - String to be checked
 - **start** (optional) - Beginning position where **prefix** is to be checked within the string.
 - **end** (optional) - Ending position where **prefix** is to be checked within the string.
- It returns **True** if the string **starts with** the specified **prefix** , otherwise returns **False**



String Comparison Methods

Example:

```
text = "Python is easy to learn."
```

```
result = text.startswith('is easy')  
print(result)
```

```
result = text.startswith('Python is ')  
print(result)
```

```
result = text.startswith('Python is easy to learn.')  
print(result)
```

```
result = text.startswith('is easy',7)  
print(result)
```

Output:

False

True

True

True



String Comparison Methods



- **endswith()**
- The **endswith()** method takes maximum of **three** parameters:
 - **suffix** - String to be checked
 - **start** (optional) - Beginning position where **suffix** is to be checked within the string.
 - **end** (optional) - Ending position where **suffix** is to be checked within the string.
- It returns **True** if the string **ends with** the specified **suffix**, otherwise returns **False**



String Comparison Methods

Example:

```
text = "Python is easy to learn."
```

```
result = text.endswith('to learn')
print(result)
```

```
result = text.endswith('to learn.')
print(result)
```

```
result = text.endswith('learn.', 7)
print(result)
```

```
result = text.endswith('is', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 14)
print(result)
```

Output:

False

True

True

False

False

True



String Searching Methods



- **index()**

Returns the **index** of **first occurrence** of a **substring** inside the string (if found).

If the substring is not found, it raises an **exception**.

Syntax: The **index()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



String Searching Methods



Example:

```
text= 'Sunday is a fun day'
```

```
result = text.index('is a fun')
print(result)
```

```
result = text.index('day')
print(result)
```

```
result = text.index('day',7)
print(result)
```

```
result = text.index('night')
print(result)
```

Output:

7

3

16

ValueError



String Searching Methods



- **find()**

Returns the **first index** of a **substring** inside the string (if found).

If the substring is not found, it returns **-1**

Syntax: The **find()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



String Searching Methods

Example:

```
text= 'Sunday is a fun day'
```

```
result = text.find('is a fun')
print(result)
```

```
result = text.find('day')
print(result)
```

```
result = text.find('day',7)
print(result)
```

```
result = text.find('night')
print(result)
```

Output:

7
3
16
-1



String Searching Methods



- **count()**

Returns the **number of occurrences** of a **substring** in the given **string**

If the substring is not found, it returns 0

Syntax: The **count()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



String Searching Methods

Example:

```
text = "Python is awesome, isn't it?"
```

```
substring = "is"
```

```
count = text.count(substring)
```

```
print(count)
```

```
substring = "i"
```

```
count = text.count(substring, 8, 25)
```

```
print(count)
```

```
substring = "ton"
```

```
count = text.count(substring)
```

```
print(count)
```

Output:

2

1

0



String Replacement Methods

- **replace()**

Returns a copy of the string where all occurrences of a **substring** is replaced with **another substring**.

Syntax: The **replace()** method takes **three** parameters:

- **old** - old substring we want to replace
 - **new** - new substring which would replace the old substring
 - **count** (optional) - the number of times we want to replace the old substring with the new substring
-
- The **replace()** method returns a copy of the string where **old substring** is replaced with the **new substring**. The original string is unchanged.
 - If the **old substring** is not found, it returns the copy of the original string.



String Replacement Methods

Example:

```
text = "Blue Blue Blue"  
newtext= text.replace("ue","ack")  
print(newtext)  
  
newtext= text.replace("ue","ack",2)  
print(newtext)  
  
newtext= text.replace("eu","ack")  
print(newtext)
```

Output:

```
Black Black Black  
Black Black Blue  
Blue Blue Blue
```



String Replacement Methods



- **strip()**

Returns a copy of the string with both **leading** and **trailing** characters removed (based on the string argument passed).

Syntax: The **strip()** method takes **one** optional parameter:

- **chars** (optional) - a string specifying the set of characters to be removed.
- If the **chars** argument is not provided, all leading and trailing whitespaces are removed from the string.



String Replacement Methods

Example:

```
text = " Good Morning"  
newtext= text.strip()  
print("Original text:[ "+text+"]")  
print("New text:[ "+newtext+"]")
```

Output:

```
Original text:[ Good Morning]  
New text:[Good Morning]
```



Exercise

Write a program to simulate user registration process. Your code should do the following:

- 1. It should first ask the user to input his full name. If he doesn't enter his full name then program should display the error message and again ask the user to enter full name . Repeat the process until the user types his full name.[full name means a string with atleast 2 words separated with a space]**
- 2. Then it should ask the user to input his password. The rules for password are:**
 - 1. It should contain atleast 8 characters**
 - 2. It should contain atleast 1 digit and 1 upper case letter**

Repeat the process until the user correctly types his Password.

Finally , display the user's first name with a THANK YOU message. Create separate functions for accepting fullname , password and returning firstname



Sample Output

```
Type your full name:Sachin
Please enter your full name!
Type your full name:Sachin Kapoor
Type your password:Admin
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Administrator1
Hello Sachin
Thank you for joining us!
```



Solution

```
def get_full_name():
    pass
def get_password():
    pass
def get_first_name(fullname):
    pass
fullname=get_full_name()
pwd=get_password()
firstname=get_first_name(fullname)
print("Hello",firstname,"\\nThank you for joining us!")
```



String Replacement Methods

- **split()**

The **split()** method breaks up a string at the specified separator and returns a list of strings.

Syntax: The **split()** method takes **two** parameters:

- **separator** (optional)- This is the **delimiter**. The string splits at the specified **separator**. If the **separator** is not specified, any whitespace (space, newline etc.) string is a **separator**.
- **maxsplit** (optional) - The **maxsplit** defines the maximum number of splits. The default value of **maxsplit** is **-1**, meaning, no limit on the number of splits.

The **split()** breaks the string at the separator and returns a list of strings.



String Replacement Methods

Example:

```
text= 'Live and let live'  
print(text.split())  
grocery = 'Milk, Butter, Bread'  
print(grocery.split(','))  
print(grocery.split(':'))
```

Output:

```
['Live', 'and', 'let', 'live']  
['Milk', ' Butter', ' Bread']  
['Milk, Butter, Bread']
```

Exercise

Write a program to which takes a **string from the user and displays a **list** of all of the **words** that the string contains **removing all special characters except **space****. It should also display **number of words** in the string.**

For example:

Input:

Let's learn C++, Python and then Java.

Output:

['Lets', 'learn', 'C', 'Python', 'and', 'then', 'Java']



String Replacement Methods

- **join()**

The **join()** method returns a **string** concatenated with the elements of an **iterable**.

But the **iterable** should only contain strings

The **join()** method takes an **iterable** - objects capable of returning its members one at a time

Syntax: `<str>.join(<iterable>)`



String Replacement Methods

Example:

```
mylist = ["C","C++","Java","Python"]
```

```
s = "->"
```

```
print(s.join(mylist))
```

Output:

C->C++->Java->Python



String Replacement Methods

Example:

```
letters = 'PYTHON'  
letters_spaced = ''.join(letters)  
print(letters_spaced)
```

Output:

P Y T H O N



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 35



Today's Agenda



• **Dictionary-I**

- What Is A Dictionary ?
- What Is Key-Value Pair ?
- Creating A Dictionary
- Important Characteristics Of A Dictionary
- Different Ways To Access A Dictionary
- An Important Point



What Is A Dictionary ?



- Python **dictionary** is an **unordered collection** of **items**.
- The collections we have studied till now like **list** , **tuple** and **string** are all **ordered collections** as well as can hold only one value as their element
- On the other hand **dictionary** is an **unordered collection** which holds the data in a **key: value** pair.



What Is Key-Value Pair?



- Sometimes we need to store the data so that one piece of information is **connected** to another piece of information.
- For example **RollNo**→**Student Name** or **Customer Name**→**Mobile Number**
- In these examples **RollNo** will be called a **key** while it's associated **Student Name** will be called **value**
- To store such paired data **Python** provides us the data type called **dictionary**



How To Create A Dictionary?



- Creating a dictionary is as simple as placing **items** inside **curly braces { }** separated by **comma**.
- Every **item** has a **key** and the corresponding **value** expressed as a **pair, key: value**.
- While **values** can be of **any data type** and **can repeat**, but **keys** must be of **immutable type** and must be **unique**.

General Syntax Of Creating A Dictionary



Syntax:

```
d = {  
    <key>: <value>,  
    <key>: <value>,  
    ...  
    <key>: <value>  
}
```



How To Create A Dictionary?



```
# empty dictionary  
my_dict = {}
```

```
# dictionary with integer keys  
my_dict = {1: 'Amit', 2: 'Brajesh', 3:'Chetan'}
```

```
# dictionary with mixed keys  
my_dict = {1: 'John', 'a':'vowel'}
```

```
# dictionary with list as values  
my_dict = {'Rahul':['C', 'C++'], 'Ajay':['Java', 'C', 'Python'],  
          'Neeraj':['Oracle', 'Python']}
```

Important Characteristics Of Dictionaries



- The important characteristics of **Python dictionaries** are as follows:
 - They can be **nested**.
Click to add text
 - They are **mutable**.
 - They are **dynamic**.
 - They are **unordered**.
 - Unlike **Lists** and **tuples**, a **dictionary item is accessed by its corresponding key not index**

Other Ways Of Creating Dictionary



- We also can create a list by using the **dict()** function

```
# Create an empty dictionary
```

```
my_dict = dict()
```

```
# Create a dictionary with elements
```

```
my_dict = dict({1:'apple', 2:'ball'})
```

```
# Create a dictionary with other sequences
```

```
my_dict = dict([(1,'apple'), (2,'ball')])
```



PYTHON

LECTURE 35

These Notes Have Python _ world _ In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Today's Agenda



- **Strings -III**
 - String Methods



String Methods



- A string object has a number of **method** or **member functions**.
- These can be grouped into different categories .
- These categories are:
 - **String conversion methods**
 - **String comparison methods**
 - **String searching methods**
 - **String replace methods**



String Conversion Methods

- **capitalize()**

Returns a copy of the string with **first character capitalized** and rest of the characters in **lower case**.

Example:

```
name="guido van rossum"  
newname=name.capitalize()  
print(f'Original name is {name}\nCapitalized name is {newname}')
```

Output:

```
Original name is guido van rossum  
Capitalized name is Guido van rossum
```



String Conversion Methods

```
name="Guido Van Rossum"  
newname=name.capitalize()  
print(f'Original name is {name}\nCapitalized name is {newname}')
```

Output:

```
Original name is Guido Van Rossum  
Capitalized name is Guido van rossum
```



String Conversion Methods

```
text="python is awesome. java rocks"  
newtext=text.capitalize()  
print(f'Original text is {text}\nCapitalized text is {newtext}')
```

Output:

```
Original text is python is awesome. java rocks  
Capitalized text is Python is awesome. java rocks
```



String Conversion Methods

- **lower()** and **upper()**

Returns a copy of the string with all letters converted to **lowercase** and **uppercase** respectively

Example:

```
name="Sachin Kapoor"  
lc=name.lower()  
uc=name.upper() print(f'Original  
name is {name}') print(f'Lower  
name is {lc}') print(f'Upper name  
is {uc}')
```

Output:

```
Original name is Sachin Kapoor  
Lower name is sachin kapoor  
Upper name is SACHIN KAPOOR
```



String Conversion Methods



- **swapcase()**

Returns a copy of the string with the **case** of every character **swapped**. Means that **lowercase characters** are changed to **uppercase** and **vice-versa**.

Example:

```
name="Sachin Kapoor"  
newname=name.swapcase()  
print(f'Original name is {name}')  
print(f'Swapped name is {newname}')
```

Output:

```
Original name is Sachin Kapoor  
Swapped name is SACHIN kAPOOR
```



String Conversion Methods



- **title()**

Returns a copy of the string converted to **proper case** or **title case**. i.e., all **words** begin with **uppercase letter** and the **rest** are in **lowercase**.

Example:

```
text="we got independence in 1947"
newtext=text.title() print(f"Original
text is {text}") print(f"Title text is
{newtext}")
```

Output:

```
Original text is we got independence in 1947
Title text is We Got Independence In 1947
```



String Conversion Methods



```
text = "i LOVE pYTHON"  
print(text.title())
```

Output:

I Love Python



String Conversion Methods

```
text = "physics,chemistry,maths"  
print(text.title())
```

Output:

```
Physics,chemistry,Maths
```



String Conversion Methods

```
text = "physics_chemistry_maths"  
print(text.title())
```

Output:

```
Physics_Chemistry_Maths
```



String Conversion Methods

```
text = "physics1chemistry2maths"  
print(text.title())
```

Output:

```
Physics1Chemistry2Maths
```



String Conversion Methods

```
text = "He's an engineer, isn't he?"  
print(text.title())
```

Output:

```
He'S An Engineer, Isn'T He?
```



String Comparison Methods

- **islower() and isupper()**

Returns **True** or **False** depending on whether all alphabets in the string are in **lowercase** and **uppercase** respectively



String Comparison Methods



Example:

```
s = 'this is good'
```

```
print(s.islower())
```

```
s = 'th!s is a1so g00d'
```

```
print(s.islower())
```

```
s = 'this is Not good'
```

```
print(s.islower())
```

Output:

```
True  
True  
False
```



String Comparison Methods



Example:

```
s = "THIS IS GOOD!"
```

```
print(s.isupper())
```

```
s = "THIS IS ALSO GooD!"
```

```
print(s.isupper())
```

```
s= "THIS IS not GOOD!"
```

```
print(s.isupper())
```

Output:

```
True  
True  
False
```



String Comparison Methods



Example:

```
s = ""
```

```
print(s.isupper())
print(s.islower())
```

Output:

```
False
False
```



String Comparison Methods



- **istitle()**

Returns **True** if the string is in **titlecase** or **empty** ,
otherwise returns **False**



String Comparison Methods

Example:

```
s = 'Python Is Good.'  
print(s.istitle())
```

```
s = 'Python is good'  
print(s.istitle())
```

```
s = 'This Is @ Symbol.'  
print(s.istitle())
```

```
s = '99 Is A Number'  
print(s.istitle())
```

```
s = 'PYTHON'  
print(s.istitle())
```

Output:

True
False
True
True
False



String Comparison Methods



- **isalpha()**

Returns **True** if the string contains only **alphabets**,
otherwise returns **False**



String Comparison Methods

Example:

```
name = "Monalisa"  
print(name.isalpha())
```

```
name = "M0nalisa"  
print(name.isalpha())
```

```
name = "Monalisa Shah"  
print(name.isalpha())
```

Output:

True
False
False



String Comparison Methods



- **isdigit()**

Returns **True** if the string contains only **digits** , otherwise returns **False**



String Comparison Methods

Example:

```
text = "12345"
```

```
print(text.isdigit())
```

```
text = "012345"
```

```
print(text.isdigit())
```

```
text = "12345 6"
```

```
print(text.isdigit())
```

```
text = "a12345"
```

```
print(text.isdigit())
```

Output:

True

True

False

False



String Comparison Methods



- **isdecimal()**

Returns **True** if the string contains only **decimal characters**, otherwise returns **False**



String Comparison Methods

Example:

```
text = "12345"
```

```
print(text.isdecimal())
```

```
text = "012345"
```

```
print(text.isdecimal())
```

```
text = "12345 6"
```

```
print(text.isdecimal())
```

```
text = "a12345"
```

```
print(text.isdecimal())
```

Output:

True

True

False

False



String Comparison Methods



- **isnumeric()**

Returns **True** if the string contains only
numeric characters, otherwise returns **False**



String Comparison Methods

Example:

```
text = "12345"
```

```
print(text.isnumeric())
```

```
text = "012345"
```

```
print(text.isnumeric())
```

```
text = "12345 6"
```

```
print(text.isnumeric())
```

```
text = "a12345"
```

```
print(text.isnumeric())
```

Output:

True

True

False

False

isdigit() V/s isdecimal() V/s isnumeric()



- To understand the difference between **isdigit()** , **isdecimal()** and **isnumeric()** , we will first have to understand what **Python** considers as **digits** , **decimal** or **numerics** for **special symbols**
- In Python:
 - **superscript** and **subscripts** (usually written using unicode) are also considered **digit** characters , **numeric** characters but not **decimals**.
 - **roman numerals**, **currency numerators** and **fractions** (usually written using unicode) are considered **numeric** characters but not **digits** or **decimals**



String Comparison Methods

Example:

```
s = '\u00B23455'  
print(s)  
print(s.isdigit())  
print(s.isdecimal())  
print(s.isnumeric())  
  
s = '\u00BD'  
print(s)  
print(s.isdigit())  
print(s.isdecimal())  
print(s.isnumeric())
```

Output:

```
'²3455'  
isdigit(): True  
isdecimal(): False  
isnumeric(): True  
½  
isdigit(): False  
isdecimal(): False  
isnumeric(): True
```



String Comparison Methods



- **isalnum()**

Returns **True** if the string contains only **alphanumeric** characters , otherwise returns **False**



String Comparison Methods

Example:

```
name = "M234onalisa"  
print(name.isalnum())
```

```
name = "M3ona Shah "  
print(name.isalnum())
```

```
name = "Mo3nalisaSha22ah"  
print(name.isalnum())
```

```
name = "133"  
print(name.isalnum())
```

Output:

True
False
True
True



String Comparison Methods



- **isspace()**

Returns **True** if the string contains only **whitespace** characters , otherwise returns **False**



String Comparison Methods

Example:

```
s = ' \t'  
print(s.isspace())
```

```
s = ' a '  
print(s.isspace())
```

```
s = ''  
print(s.isspace())
```

```
s = "  
print(s.isspace())
```

Output:

True
False
True
False



String Comparison Methods



- **startswith()**
- The **startswith()** method takes maximum of **three** parameters:
 - **prefix** - String to be checked
 - **start** (optional) - Beginning position where **prefix** is to be checked within the string.
 - **end** (optional) - Ending position where **prefix** is to be checked within the string.
- It returns **True** if the string **starts with** the specified **prefix** , otherwise returns **False**



String Comparison Methods

Example:

```
text = "Python is easy to learn."
```

```
result = text.startswith('is easy')  
print(result)
```

```
result = text.startswith('Python is ')  
print(result)
```

```
result = text.startswith('Python is easy to learn.')  
print(result)
```

```
result = text.startswith('is easy',7)  
print(result)
```

Output:

False

True

True

True



String Comparison Methods



- **endswith()**
- The **endswith()** method takes maximum of **three** parameters:
 - **suffix** - String to be checked
 - **start** (optional) - Beginning position where **suffix** is to be checked within the string.
 - **end** (optional) - Ending position where **suffix** is to be checked within the string.
- It returns **True** if the string **ends with** the specified **suffix**, otherwise returns **False**



String Comparison Methods

Example:

```
text = "Python is easy to learn."
```

```
result = text.endswith('to learn')
print(result)
```

```
result = text.endswith('to learn.')
print(result)
```

```
result = text.endswith('learn.', 7)
print(result)
```

```
result = text.endswith('is', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 14)
print(result)
```

Output:

False

True

True

False

False

True



String Searching Methods



- **index()**

Returns the **index** of **first occurrence** of a **substring** inside the string (if found).

If the substring is not found, it raises an **exception**.

Syntax: The **index()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



String Searching Methods



Example:

```
text= 'Sunday is a fun day'
```

```
result = text.index('is a fun')
print(result)
```

```
result = text.index('day')
print(result)
```

```
result = text.index('day',7)
print(result)
```

```
result = text.index('night')
print(result)
```

Output:

7

3

16

ValueError



String Searching Methods



- **find()**

Returns the **first index** of a **substring** inside the string (if found).

If the substring is not found, it returns **-1**

Syntax: The **find()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



String Searching Methods



Example:

```
text= 'Sunday is a fun day'
```

```
result = text.find('is a fun')
print(result)
```

```
result = text.find('day')
print(result)
```

```
result = text.find('day',7)
print(result)
```

```
result = text.find('night')
print(result)
```

Output:

7
3
16
-1



String Searching Methods



- **rfind()**

Returns the **highest index** of a **substring** inside the string (if found).

If the substring is not found, it returns -1

Syntax: The **rfind()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



String Searching Methods



Example:

```
text= 'Sunday is a fun day'
```

```
result = text.rfind('is a fun')
print(result)
```

```
result = text.rfind('day')
print(result)
```

```
result = text.rfind('day',0)
print(result)
```

```
result = text.rfind('night')
print(result)
```

Output:

7

16

16

-1



String Searching Methods



- **count()**

Returns the **number of occurrences** of a **substring** in the given **string**

If the substring is not found, it returns 0

Syntax: The **count()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



String Searching Methods

Example:

```
text = "Python is awesome, isn't it?"
```

```
substring = "is"
```

```
count = text.count(substring)
```

```
print(count)
```

```
substring = "i"
```

```
count = text.count(substring, 8, 25)
```

```
print(count)
```

```
substring = "ton"
```

```
count = text.count(substring)
```

```
print(count)
```

Output:

2

1

0



String Replacement Methods

- **replace()**

Returns a copy of the string where all occurrences of a **substring** is replaced with **another substring**.

Syntax: The **replace()** method takes **three** parameters:

- **old** - old substring we want to replace
 - **new** - new substring which would replace the old substring
 - **count** (optional) - the number of times we want to replace the old substring with the new substring
-
- The **replace()** method returns a copy of the string where **old substring** is replaced with the **new substring**. The original string is unchanged.
 - If the **old substring** is not found, it returns the copy of the original string.



String Replacement Methods

Example:

```
text = "Blue Blue Blue"  
newtext= text.replace("ue","ack")  
print(newtext)  
  
newtext= text.replace("ue","ack",2)  
print(newtext)  
  
newtext= text.replace("eu","ack")  
print(newtext)
```

Output:

```
Black Black Black  
Black Black Blue  
Blue Blue Blue
```



String Replacement Methods



- **strip()**

Returns a copy of the string with both **leading** and **trailing** characters removed (based on the string argument passed).

Syntax: The **strip()** method takes **one** optional parameter:

- **chars** (optional) - a string specifying the set of characters to be removed.
- If the **chars** argument is not provided, all leading and trailing whitespaces are removed from the string.



String Replacement Methods

Example:

```
text = " Good Morning"  
newtext= text.strip()  
print("Original text:[ "+text+"]")  
print("New text:[ "+newtext+"]")
```

Output:

```
Original text:[ Good Morning]  
New text:[Good Morning]
```



Exercise

Write a program to simulate user registration process. Your code should do the following:

- 1. It should first ask the user to input his full name. If he doesn't enter his full name then program should display the error message and again ask the user to enter full name . Repeat the process until the user types his full name.[full name means a string with atleast 2 words separated with a space]**
- 2. Then it should ask the user to input his password. The rules for password are:**
 - 1. It should contain atleast 8 characters**
 - 2. It should contain atleast 1 digit and 1 upper case letter**

Repeat the process until the user correctly types his Password.

Finally , display the user's first name with a THANK YOU message. Create separate functions for accepting fullname , password and returning firstname



Sample Output

```
Type your full name:Sachin
Please enter your full name!
Type your full name:Sachin Kapoor
Type your password:Admin
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Administrator1
Hello Sachin
Thank you for joining us!
```



Solution



```
def get_full_name():
    while True:
        name=input("Type your full name:").strip()
        if name.find(" ")!=-1:
            return name
        print("Please enter your full name!")

def get_password():
    while True:
        pwd=input("Type your password:")
        cap_letter_present=False
        digit_present=False
        for x in pwd:
            if x.isdigit():
                digit_present=True
            elif x.isupper():
                cap_letter_present=True
        if digit_present==False or cap_letter_present==False or len(pwd)<8:
            print("Password must be of 8 or more characters in length\nwith atleast 1 digit and 1 capital letter")
        else:
            return pwd

def get_first_name(fullname):
    spacepos=fullname.find(" ")
    return fullname[0:spacepos]

fullname=get_full_name()
pwd=get_password()
firstname=get_first_name(fullname)
print("Hello",firstname,"\\nThank you for joining us!")
```



String Replacement Methods

- **split()**

The **split()** method breaks up a string at the specified separator and returns a list of strings.

Syntax: The **split()** method takes **two** parameters:

- **separator** (optional)- This is the **delimiter**. The string splits at the specified **separator**. If the **separator** is not specified, any whitespace (space, newline etc.) string is a **separator**.
- **maxsplit** (optional) - The **maxsplit** defines the maximum number of splits. The default value of **maxsplit** is **-1**, meaning, no limit on the number of splits.

The **split()** breaks the string at the separator and returns a list of strings.



String Replacement Methods

Example:

```
text= 'Live and let live'  
print(text.split())  
grocery = 'Milk, Butter, Bread'  
print(grocery.split(','))  
print(grocery.split(':'))
```

Output:

```
['Live', 'and', 'let', 'live']  
['Milk', 'Butter', 'Bread']  
['Milk,Butter,Bread']
```



String Replacement Methods

- **join()**

The **join()** method returns a **string** concatenated with the elements of an **iterable**.

But the **iterable** should only contain strings

The **join()** method takes an **iterable** - objects capable of returning its members one at a time

Syntax: `<str>.join(<iterable>)`



String Replacement Methods

Example:

```
mylist = ["C","C++","Java","Python"]
```

```
s = "->"
```

```
print(s.join(mylist))
```

Output:

C->C++->Java->Python



String Replacement Methods

Example:

```
letters = 'PYTHON'  
letters_spaced = ''.join(letters)  
print(letters_spaced)
```

Output:

P Y T H O N



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 36



Today's Agenda



- **Dictionary-II**
 - Updating Elements In Dictionary
 - Removing Elements From Dictionary
 - Functions Used In Dictionary



Updating A Dictionary



- Since dictionary is mutable, so we can **add new items** or **change the value** of **existing items** using either of two ways.
- These are:
 - **assignment operator** or
 - **update()** method of **dictionary** object

Updating Using Assignment Operator



- **Syntax Of Assignment Operator:**

`dict_var[key]=value`

- When we use **assignment operator** , Python simply searches for the **key** in the **dictionary object**.
- If the **key** is found , it's **value** is replaced with the **value** we have passed, otherwise a ***new key-value pair entry is created***



Guess The Output ?

```
student_data = {1:'Amit', 2:'Brajesh',3:'Chetan',
4:'Deepak',5:'Neeraj'}
print("Before updating:")
print(student_data)
student_data[2]='Brajendra'
print("After updating:")
print(student_data)
```

Output:

```
Before updating:
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}
After updating:
{1: 'Amit', 2: 'Brajendra', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}
```



Guess The Output ?

```
student_data = {1:'Amit', 2:'Brajesh',3:'Chetan',  
 4:'Deepak',5:'Neeraj'}  
print("Before updating:")  
print(student_data)  
student_data[8]='Ankit'  
print("After updating:")  
print(student_data)
```

Output:

```
Before updating:  
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}  
After updating:  
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj', 8: 'Ankit'}
```

Updating Using **update()** Method



- **Syntax Of update() Method:**

dict_var.update(dict_var2)

- The **update()** method **merges** the **keys** and **values** of one **dictionary/iterable** into another, **overwriting** values of the **same key**



Guess The Output ?

```
student_data = {1:'Amit', 2:'Brajesh',3:'Chetan',
4:'Deepak',5:'Neeraj'}
student_data2={2:'Brajendra',8:'Ankit'}
print("Before updating:")
print(student_data)
student_data.update(student_data2)
print("After updating:")
print(student_data)
```

Output:

```
Before updating:
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}
After updating:
{1: 'Amit', 2: 'Brajendra', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj', 8: 'Ankit'}
```



Exercise

Write a program to create a dictionary called **accounts containing **account id** and **balance** of account holders . Initialize it with the following data:**

101: 50000

102:45000

103:55000

Now ask the user to input an **account id and **amount** . If the **account id** is present in the dictionary then **update** the **balance** by adding the **amount** given otherwise add a new entry of **account id** and **balance** in the dictionary. Finally print all the **accounts** details.**



Sample Output



```
Current Accounts Present
{101: 50000, 102: 45000, 103: 55000}
Enter account id:101
Enter amount:4000
Account updated
Account Details:
{101: 54000, 102: 45000, 103: 55000}
```

```
Current Accounts Present
{101: 50000, 102: 45000, 103: 55000}
Enter account id:104
Enter amount:60000
New Account Created
Account Details:
{101: 50000, 102: 45000, 103: 55000, 104: 60000}
```

Solution

```
accounts={101:50000,102:45000,103:55000}
print("Current Accounts Present")
print(accounts)
id=int(input("Enter account id:"))
amt=int(input("Enter amount:"))
balance=accounts.get(id)
if balance == None:
    accounts[id]=amt
    print("New Account Created")
else:
    newbalance=balance+amt
    accounts[id]=newbalance
    print("Account updated")
print("Account Details:")
print(accounts)
```

These Notes Have Python_world_India. So commercial Use of this notes is Strictly prohibited

Removing Data From Dictionary



- Since dictionary is **mutable**, so we can **remove items** from the dictionary.
- There are certain **methods** available in **Python** to **delete** the **items** or **delete entire** dictionary.
- These are:
 - **pop(key)**: removes the entry with provided **key** and returns its **value**
 - **del**: deletes a single item or the dictionary entirely
 - **clear()**: clears all the items in the dictionary and returns an empty dictionary



The `pop()` Method

- Syntax Of `pop()` Method:

`dict_var.pop(key,[default])`

- The `pop()` method takes **two** parameters:
 - **key** - key which is to be searched for removal
 - **default** - (optional) value which is to be returned when the key is not in the dictionary
- Return Value From `pop()`
 - If **key** is found – it returns **value** of removed/popped element from the dictionary
 - If **key** is not found – it returns the value specified as the **second argument (default)**
 - If key is not found and default argument is not specified – it raises **KeyError** exception



Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(sixMonths)
```

```
print(sixMonths.pop("Jun"))
```

```
print(sixMonths)
```

```
print(sixMonths.pop("Jul",-1))
```

```
print(sixMonths.pop("Jul","Not found"))
```

```
print(sixMonths.pop("Jul"))
```

Output:

```
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30}
```

```
30
```

```
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31}
```

```
-1
```

```
Not found
```

```
Traceback (most recent call last):
```

```
  File "dictdemo4.py", line 8, in <module>  
    print(sixMonths.pop("Jul"))
```

```
KeyError: 'Jul'
```



The **del** Operator

- Just like **list** , **Python** also allows us to **delete** an item from the dictionary by calling the **operator/keyword del**
- **Syntax Of `del` Operator:**

`del dict_var[key]`

- It removes the **entry** from the dictionary whose **key** is passed as argument
- If the **key** is **not found** or **dictionary** is **empty** it raises **KeyError** exception
- If we do not pass the key then **del** deletes the **entire dictionary object**



Guess The Output ?

```
threeMonths = {"Jan":31, "Feb":28, "Mar":31}  
print(threeMonths)  
del threeMonths  
print(threeMonths)
```

Output:

```
{'Jan': 31, 'Feb': 28, 'Mar': 31}  
Traceback (most recent call last):  
  File "dictdemo4.py", line 5, in <module>  
    print(threeMonths)  
NameError: name 'threeMonths' is not defined
```



The **clear()** Method



- The **clear()** method **removes all items** from the dictionary.
- **Syntax Of clear() Method:**

dict_var.clear()

- The **clear()** method doesn't take any argument
- It returns nothing (None)



Guess The Output ?

```
threeMonths = {"Jan":31, "Feb":28, "Mar":31}  
print(threeMonths)  
threeMonths.clear()  
print(threeMonths)
```

Output:

```
{'Jan': 31, 'Feb': 28, 'Mar': 31}  
}
```

Functions Used With Dictionary



- Like **list** , Python allows us to use the following functions with **dictionary** object
 - **len()**
 - **max()**
 - **min()**
 - **any()**
 - **all()**
 - **sorted()**



The **len()** Function

- Returns the **number of items** in the dictionary

Example:

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(len(sixMonths))
```

Output:

6



The **max()** Function

- Returns the **greatest key** present in the dictionary

Example:

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(max(sixMonths))
```

Output:

May



Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, 3:31, "Apr":30, 5:31, 6:30}  
print(max(sixMonths))
```

Output:

```
print(max(sixMonths))  
TypeError: '>' not supported between instances of 'int' and 'str'
```



Guess The Output ?

```
sixMonths = {1:31, 2:28, 3:31,4:30,5:31,6:30}  
print(max(sixMonths))
```

Output:

6



Guess The Output ?



```
data = {False: 10, True: 5}  
print(max(data))
```

Output:

True



Guess The Output ?



```
data={False:0,True:1,None:2}  
print(max(data))
```

Output:

```
print(max(data))  
TypeError: '>' not supported between instances of 'NoneType' and 'bool'
```



Exercise



Write a program to create a dictionary called `players` and accept names of `5 players` and their `runs` from the user. Now find out the `highest score`



Sample Output

```
Enter player name:Virat
Enter runs:100
Enter player name:Dhoni
Enter runs:200
Enter player name:Shikhar
Enter runs:45
Enter player name:Kedar
Enter runs:50
Enter player name:Raina
Enter runs:75
{'Virat': 100, 'Dhoni': 200, 'Shikhar': 45, 'Kedar': 50, 'Raina': 75}
Highest runs are : 200
```



Solution

```
players={}
i=1
while i<=5:
    name=input("Enter player name:")
    runs=int(input("Enter runs:"))
    players[name]=runs
    i=i+1
print(players)
runs=max(players.values())
print("Highest runs are :",runs)
```



Exercise



Modify the previous code so that you are able to find out the name of the player also who has scored the highest score



Sample Output

```
Enter player name:Virat
Enter runs:100
Enter player name:Shikar
Enter runs:45
Enter player name:Dhoni
Enter runs:120
Enter player name:Kedar
Enter runs:90
Enter player name:Raina
Enter runs:50
{'Virat': 100, 'Shikar': 45, 'Dhoni': 120, 'Kedar': 90, 'Raina': 50}
Player with top score is Dhoni with score of 120
```



Solution

```
players={}
i=1
while i<=5:
    name=input("Enter player name:")
    runs=int(input("Enter runs:"))
    players[name]=runs
    i=i+1
print(players)
max=0
pl=""
for name,runs in players.items():
    if runs>max:
        max=runs
        pl=name
print("Player with top score is",pl,"with score of",max)
```



The **min()** Function



- Returns the **least** item present in the dictionary

Example:

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(min(sixMonths))
```

Output:

Apr



The **any()** Function

- Like list and tuple , **any()** function accepts a **dict** as argument and returns **True** if atleast **one element** of the **dict** is **True**. If not, this method returns **False**. If the **dict** is empty, then also it returns **False**

Example:

```
data={1:31,2:28,3:30}
```

```
print(any(data))
```

Output:

```
True
```



Guess The Output ?



```
data={0:1,False:2,"":3}  
print(any(data))
```

Output:

False



Guess The Output ?



```
data={'o':1,False:2,'':3}  
print(any(data))
```

Output:

True



Guess The Output ?



```
data= {}  
print(any(data))
```

Output:

False



The **all()** Function



- The **all()** function accepts a **dict** as argument and returns **True** if **all the keys** of the **dict** are **True** or if the **List** is **empty**. If not, this method returns **False**.

Example:

```
data={1:31,2:28,3:30,0:10}
```

```
print(all(data))
```

Output:

```
False
```



Guess The Output ?



```
data={1:31,2:28,3:30}  
print(all(data))
```

Output:

True



Guess The Output ?

```
data={'o':1,True:2,' ':3}  
print(all(data))
```

Output:

True



Guess The Output ?



```
data= {}  
print(all(data))
```

Output:

True



The **sorted()** Function



- Like it is with **lists** and **tuples**, the **sorted()** function returns a **sorted list** of only **keys** in the **dictionary**.
- The sorting is in **ascending order**, and doesn't modify the original **dictionary**.



Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
             "Jun":30}  
print(sorted(sixMonths))  
print(sixMonths)
```

Output:

```
['Apr', 'Feb', 'Jan', 'Jun', 'Mar', 'May']  
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30}
```



Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30,  
             4:31, 5:30}  
  
print(sorted(sixMonths))
```

Output:

```
print(sorted(sixMonths))  
TypeError: '<' not supported between instances of 'int' and 'str'
```



Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30,  
"May":31, "Jun":30}  
print(sorted(sixMonths,reverse=True))
```

Output:

```
['May', 'Mar', 'Jun', 'Jan', 'Feb', 'Apr']
```

The **kwargs** Function Argument



- To understand **kwargs**, try to figure out the output of the code below

Example:

```
def addnos(x,y,z):
    print("sum:",x+y+z)
addnos(10,20,30)
addnos(10,20,30,40,50)
```

Output:

```
sum: 60
Traceback (most recent call last):
  File "inp_5.py", line 4, in <module>
    addnos(10,20,30,40,50)
TypeError: addnos() takes 3 positional arguments but 5 were given
```

The **kwargs** Function Argument



- To overcome this problem , we used the technique of **variable length arguments**, where we prefix the function parameter with an asterisk
- This allows us to **pass any number of arguments** to the function and inside the function they are received as **tuple**

The **args** Function Argument



```
def addnos(*args):  
    sum=0  
    for x in args:  
        sum=sum+x  
    print("sum:",sum)  
addnos(10,20,30)  
addnos(10,20,30,40,50)
```

Output

```
sum: 60  
sum: 150
```

The **args** Function Argument



- Using ***args** , we cannot pass **keyword arguments**
- So , **Python** has given us a solution for this ,called ****kwargs**, which allows us to pass the **variable length of keyword arguments** to the function.

The **kwargs** Function Argument



- In the function, we use the **double asterisk **** before the **parameter name** to denote this type of argument.
- The arguments are passed as a **dictionary** and the name of the dictionary is the name of parameter
- The **keywords** become **keys** and the **actual data** passed becomes **values**

The **kwargs** Function Argument



```
def show_details(**data):
```

```
    print("\nData type of argument:", type(data))
```

```
    for key, value in data.items():
```

```
        print("{} is {}".format(key, value))
```

```
show_details(Firstname="Sachin", Lastname="Kapoor", Age=38,  
Phone=9826012345)
```

```
show_details(Firstname="Amit", Lastname="Sharma",  
Email="amit@gmail.com", Country="India", Age=25,  
Phone=9893198931)
```

The **kwargs** Function Argument



```
Data type of argument: <class 'dict'>
Firstname is Sachin
Lastname is Kapoor
Age is 38
Phone is 9826012345
```

```
Data type of argument: <class 'dict'>
Firstname is Amit
Lastname is Sharma
Email is amit@gmail.com
Country is India
Age is 25
Phone is 9893198931
```



PYTHON

LECTURE 37

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Today's Agenda



- **Dictionary-III**
 - Dictionary Methods
 - Dictionary Comprehension
 - Restrictions On Keys



Dictionary Methods



- Python provides us following methods to work upon dictionary object:
 - **clear()**
 - **copy()**
 - **get()**
 - **items()**
 - **keys()**
 - **pop()**
 - **update()**
 - **values()**



The **copy()** Method

- This method returns a **shallow copy** of the **dictionary**.

Syntax:

`dict.copy()`

Example:

```
original = {1:'one', 2:'two'}  
new = original.copy()  
print('new:', new)  
print('original:', original)
```

Output:

```
new:  {1: 'one', 2: 'two'}  
original:  {1: 'one', 2: 'two'}
```



copy() v/s =

- When **copy()** method is used, a **new dictionary** is created which is filled with a **copy of the data** from the original dictionary.
- When **=** operator is used, a **new reference** to the **original dictionary** is created.



Guess The Output ?

```
original = {1:'one', 2:'two'}  
new = original  
new.clear()  
print('new: ', new)  
print('original: ', original)
```

Output:

```
new:  {}  
original:  {}
```

```
original = {1:'one', 2:'two'}  
new = original.copy()  
new.clear()  
print('new: ', new)  
print('original: ', original)
```

Output:

```
new:  {}  
original:  {1: 'one', 2: 'two'}
```

Using **in** And **not in** With Dictionary



- We can apply the '**in**' and '**not in**' operators on a dictionary to check whether it contains a certain **key**.
- If the **key** is **present** then **in** returns **True**, otherwise it returns **False**.
- Similarly , if the **key** is **not present** **not in** returns **True** , otherwise it returns **False**



Guess The Output ?

```
cars = {"Maruti":"Ciaz","Hyundai":"Verna","Honda":"Amaze"}  
print(cars)  
print("Hyundai is present:", "Hyundai" in cars)  
print("Audi is present:", "Audi" in cars)  
print("Renault not present:", "Renault" not in cars)
```

Output:

```
{'Maruti': 'Ciaz', 'Hyundai': 'Verna', 'Honda': 'Amaze'}  
Hyundai is present: True  
Audi is present: False  
Renault not present: True
```



Dictionary Comprehension



- Just like we have **list comprehension**, we also have **dictionary comprehension**
- **Dictionary Comprehension** is a mechanism for transforming **one dictionary** into **another dictionary**.
- During this **transformation**, items within the **original dictionary** can be **conditionally** included in the **new dictionary** and each item can be transformed as needed.

Syntax For Dictionary Comprehension



- **Syntax:**

```
dict_variable = { key:value for (key,value) in iterable}
```

- **Explanation**

- **Iterable** can be any object on which iteration is possible
- **(key,value)** is the **tuple** which will receive these **key-value** pairs one at a time
- **key:value** is the expression or **key-value** pair which will be assigned to **new dictionary**



Exercise

- Write a program to produce a **copy** of the dictionary **cars** using **dictionary comprehension**

```
cars = {"Maruti": "Ciaz", "Hyundai": "Verna", "Honda": "Amaze"}  
newcars={ k:v for (k,v) in cars.items()}  
print(newcars)
```

Output:

```
{'Maruti': 'Ciaz', 'Hyundai': 'Verna', 'Honda': 'Amaze'}
```



Exercise

- Write a program to produce a **new dictionary** from the given dictionary with the **values** of each key **getting doubled**

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}  
double_dict1 = {k:v*2 for (k,v) in dict1.items()}  
print(double_dict1)
```

Output:

```
{'a': 2, 'b': 4, 'c': 6, 'd': 8, 'e': 10}
```

Exercise

- Write a program to produce a **new dictionary** from the given dictionary with the **keys** of each key **getting doubled**

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}  
double_dict1 = {k*2:v for (k,v) in dict1.items()}  
print(double_dict1)
```

Output:

```
{'aa': 1, 'bb': 2, 'cc': 3, 'dd': 4, 'ee': 5}
```

Exercise



- Write a program to accept a string from the user and print the frequency count of it's letters , i.e. how many times each letter is occurring in the string

Output:

```
Type a string:WE LOVE INDIA
W : 1
E : 2
E : 2
L : 1
O : 1
V : 1
I : 2
N : 1
D : 1
I : 1
A : 1
```



Solution

```
str=input("Type a string:")  
mydict={ch:str.count(ch) for ch in str}  
for k,v in mydict.items():  
    print(k,":",v)
```

Adding Conditions To Dictionary Comprehension



- Like list comprehension , **dictionary comprehension** also allows us to add **conditions** to make it more powerful.

- **Syntax:**

```
dict_variable = { key:value for (key,value) in iterable <test_cond> }
```

- As usual , only those **key-value** pairs will be returned by **dictionary comprehension** which satisfy the condition

Exercise

- Write a program to produce a **new dictionary** from the given dictionary but with the **values** that are greater than **2** and store their doubles

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
dict2 = {k:v*2 for (k,v) in dict1.items() if v>2}
```

```
print(dict2)
```

Output:

```
{'c': 6, 'd': 8, 'e': 10}
```

Exercise

- Write a program to produce a **new dictionary** from the given dictionary but with the double of the **values that are greater than 2** as well as **multiple of 2**

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
dict2 = {k:v*2 for (k,v) in dict1.items() if v>2 if v%2==0}
```

```
print(dict2)
```

Output:

```
{'d': 8}
```

Exercise

- Write a program to produce a **new dictionary** from the given dictionary but the value should be the string “**EVEN**” for even values and “**ODD**” for odd values

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
dict2= {k:'even' if v%2==0 else 'odd' for (k,v) in dict1.items()}
```

```
print(dict2)
```

Output:

```
{'a': 'odd', 'b': 'even', 'c': 'odd', 'd': 'even', 'e': 'odd'}
```

Restrictions On Dictionary Keys



1. Almost any type of value can be used as a dictionary key in **Python**, like **integer**, **float**, **Boolean** etc

Example:

```
d={65:"A", 3.14:"pi", True:1}
```

```
print(d)
```

Output:

```
{65: 'A', 3.14: 'pi', True: 1}
```

Restrictions On Dictionary Keys



2. We can even use class names as keys.

Example:

```
d={int:1, float:2, bool:3}
```

```
print(d)
```

```
print(d[float])
```

Output:

```
{<class 'int'>: 1, <class 'float'>: 2, <class 'bool'>: 3}  
2
```

Restrictions On Dictionary Keys



3. Duplicate keys are not allowed. If we assign a value to an already existing dictionary key, it does not add the key a second time, but replaces the existing value:

Example:

```
d={"MP":"Indore","UP":"Lucknow","RAJ":"Jaipur"}  
print(d)  
d["MP"]="Bhopal"  
print(d)
```

Output:

```
{'MP': 'Indore', 'UP': 'Lucknow', 'RAJ': 'Jaipur'}  
{'MP': 'Bhopal', 'UP': 'Lucknow', 'RAJ': 'Jaipur'}
```

Restrictions On Dictionary Keys



4. If we specify a key a second time during the initial creation of a dictionary, the second occurrence will override the first:

Example:

```
d={"MP":"Indore","UP":"Lucknow","RAJ":"Jaipur","MP":"Bhopal"}  
print(d)
```

Output:

```
{'MP': 'Bhopal', 'UP': 'Lucknow', 'RAJ': 'Jaipur'}
```

Restrictions On Dictionary Keys



5. A dictionary key must be of a type that is **immutable**. Like **integer**, **float**, **string** and **Boolean**—can serve as dictionary keys. Even a **tuple** can also be a dictionary key, because **tuples** are **immutable**:

Example:

```
d = {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}  
print(d)  
print(d[(1,2)])
```

Output:

```
{(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}  
b
```

Restrictions On Dictionary Keys



6. However, neither a list nor another dictionary can serve as a dictionary key, because lists and dictionaries are mutable:

Example:

```
d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}  
print(d)
```

Output:

```
d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}  
TypeError: unhashable type: 'list'
```

Restrictions On Dictionary Values



- There are **no restrictions** on dictionary **values**.
- A dictionary value can be any type of object Python supports, including **mutable types** like **lists** and **dictionaries**, and **user-defined objects**
- There is also no restriction against a particular value appearing in a dictionary multiple times:



Exercise



- Write a complete **COUNTRY MANAGEMENT APP**. Your code should store **COUNTRY CODE** and **COUNTRY NAME** as **key-value** pair in a **dictionary** and allow perform following operations on the dictionary :
 - **View**
 - **Add**
 - **Delete**

To start the program initialize your dictionary with the following set of key-value pairs:

IN→India

US→America

AU→Australia

CA→Canada



Sample Output

```
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IN  
Country is India  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IT  
There is no country for country code IT  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:add
```

```
Your choice:add  
Enter country code:IT  
Enter country name:Italy  
Italy added to the list  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN IT US  
Enter country code:IT  
Country is Italy  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:add  
Enter country code:IN  
IN is already used by India  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```



Solution



```
def show_menu():
```

```
    print("SELECT AN OPTION:")
```

```
    print("view: View country names")
```

```
    print("add: Add a country")
```

```
    print("del: Delete a country")
```

```
    print("exit- Exit the program")
```

```
    print()
```

```
def show_codes(countries):
```

```
    pass
```



Solution



```
def view_country(countries):  
    pass
```

```
def add_country(countries):  
    pass
```

```
def del_country(countries):  
    pass
```



Solution

```
countries={"IN":"India","US":"America","AU":"Australia","CA  
":"Canada"}
```

while True:

show_menu()

choice=input("Your choice:")

if choice=="view":

view_country(countries)

elif choice=="add":

add_country(countries)

elif choice=="del":

del_country(countries)

elif choice=="exit":

break;

else:

print("Wrong choice ! Try again!")



Exercise



- Modify the previous code , so that now you are able to store 3 values for each key . These are COUNTRY NAME , CAPITAL CITY and POPULATION. Provide same options to the user and start with the following data

IN→India , Delhi,1320000000

US→America,Washington,320000000

AU→Australia,Canberra,24000000

CA→Canada,Ottawa,940000



Sample Output

```
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IN  
Country name is: India  
Country capital is: Delhi  
Country population is: 1320000000  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IT  
There is no country for country code IT  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:add  
Enter country code:IT  
Enter country name:Italy  
Enter capital city:Rome  
Enter population:2870000  
Italy added to the list  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN IT US  
Enter country code:IT  
Country name is: Italy  
Country capital is: Rome  
Country population is: 2870000  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:exit
```



These Notes Have Python - In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 38



Today's Agenda

• **Introduction To Object Oriented Programming-I**

- Problems With Procedure Oriented Programming
- What Is Object Oriented Programming ?
- What Is A Class ?
- What Is An Object ?
- Syntax Of Creating A Class In Python
- Syntax Of Creating Object
- Types Of Data Members A Class Can Have
- The Method **`__init__()`**
- The Argument **`self`**
- Passing Parameters To **`__init__()`**



Question ???



- Can you tell , what kind of **programming paradigm** we have followed this point in **Python** ?
- The answer is : **POP (Procedure Oriented Programming)**
 - In all the programs we wrote till now, we have designed our program around **functions** i.e. **blocks of statements which manipulate data**.
 - This is called the ***procedure-oriented programming***.



Advantages



- **Advantages Of Procedure Oriented Programming**

- It's **easy** to implement
- The ability to **re-use the same code** at different places in the program **without copying it**.
- An easier way to **keep track** of program flow **for small codes**
- Needs **less memory**.



Disadvantages

- **Disadvantages Of Procedure Oriented Programming**

- **Very difficult** to relate with **real world objects**.
- **Data is exposed** to whole program, so **no security for data**.
- **Difficult** to create **new data types**
- Importance is given to the **operation on data** rather than **the data**.



So , What Is The Solution ?



- Solution to all the previous **4 problems** is **Object Oriented Programming**
- Many people consider **OOP** to be a modern programming paradigm, but the roots go back to **1960s**.
- The **first programming language to use objects** was **Simula 67**



What Is OOP?



- **OOP** is a **programming paradigm** (*way of developing programs*)
- In **OOP**, we have the **flexibility** to represent **real-world objects** like **car**, **animal**, **person**, **ATM** etc. in our code
- It allows us to **combine** the **data** and **functionality** and **wrap it inside** something which is called an **object**



What Is An Object?



- In programming **any real world entity** which has specific **attributes** or **features** can be represented as an **object**.
- In simple words, an **object** is something that possess some **characteristics** and can **perform certain functions**.



What Is An Object?



- For example, **car** is an **object** and can perform **functions** like **start**, **stop**, **drive** and **brake**.
- These are the **functions** or **behaviours** of a car.
- And the **characteristics** or **attributes** are **color** of car, **mileage**, **maximum speed**, **model** , **year** etc.



Are We Objects ?



- Yes , **we humans** are **objects** because:
 - We have **attributes** as **name, height, age** etc.
 - We also can show **behaviors** like **walking, talking, running, eating** etc



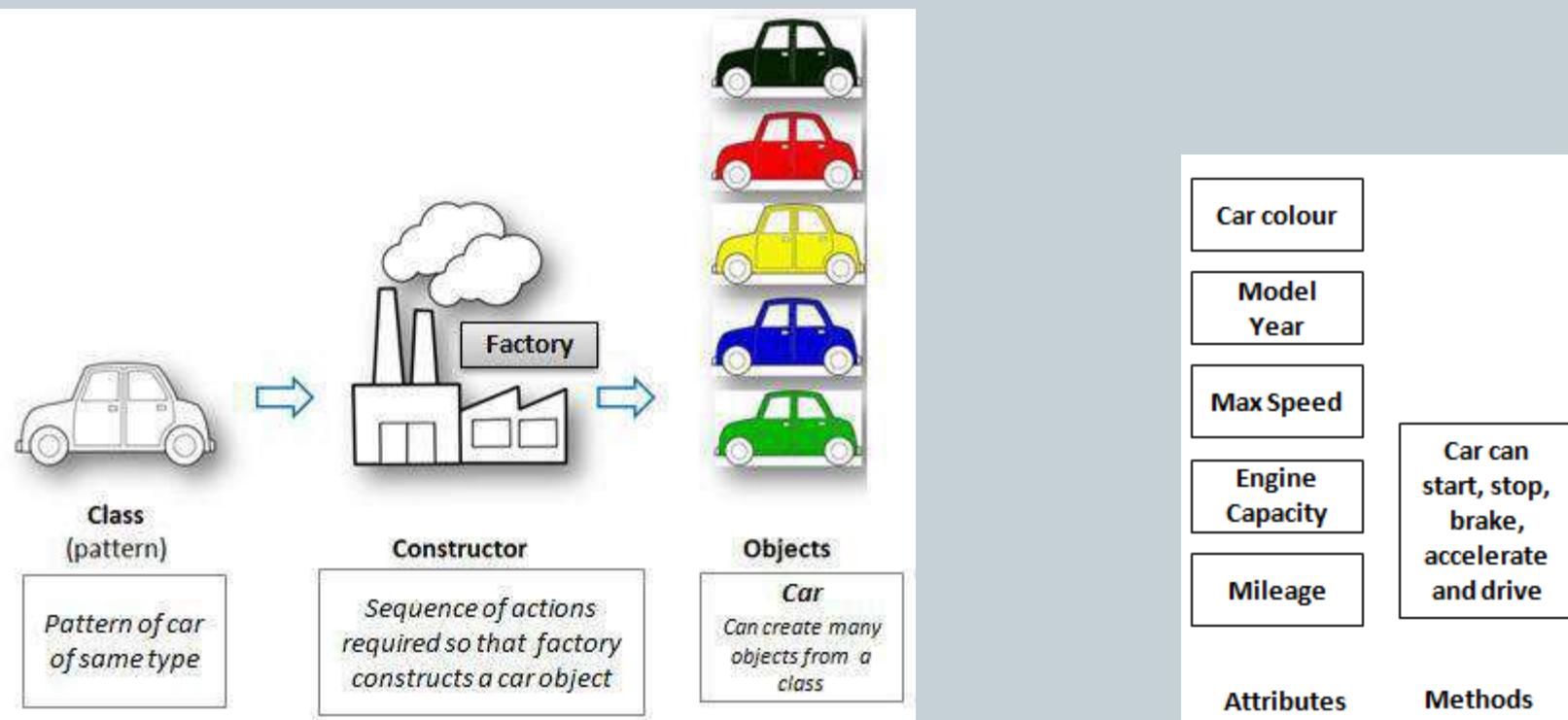
Classes



- Now to **create/represent** objects we first have to write all their **attributes** and **behaviours** under a **single group**.
- This **group** is called a **class**.
- Thus a class is an **architecture/blueprint** of the object. It is a **proper description** of the **attributes** and **methods** of the object.

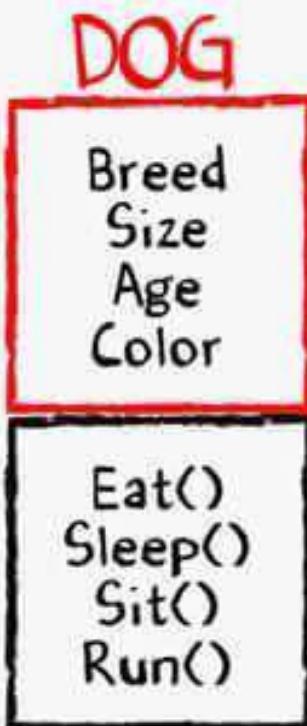
Classes

- **For Example:-** The design of a **car** of same type is a **class**. We can create **many objects** from a **class**. Just like we can make **many cars** of the same type from a **design** of **car**.





A Dog Class



Breed = Neapolitan Mastiff
Size = Large
Age = 5 years
Color = Black

Breed = Maltese
Size = Small
Age = 2 years
Color = White

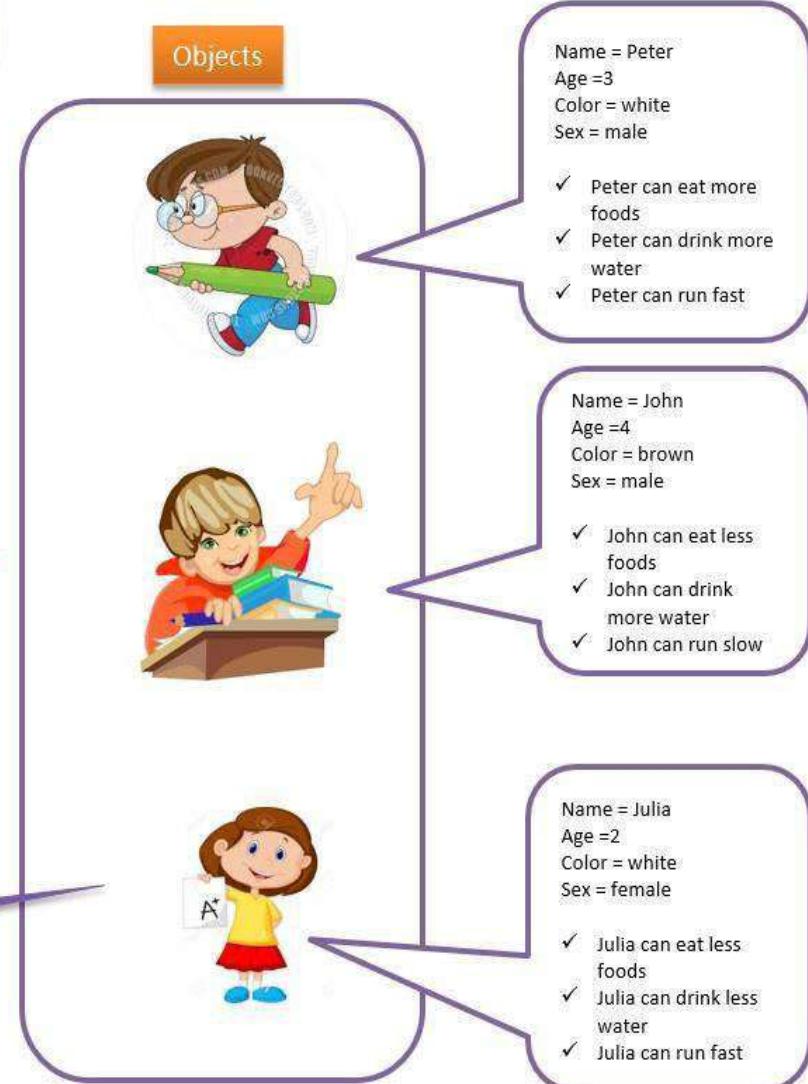
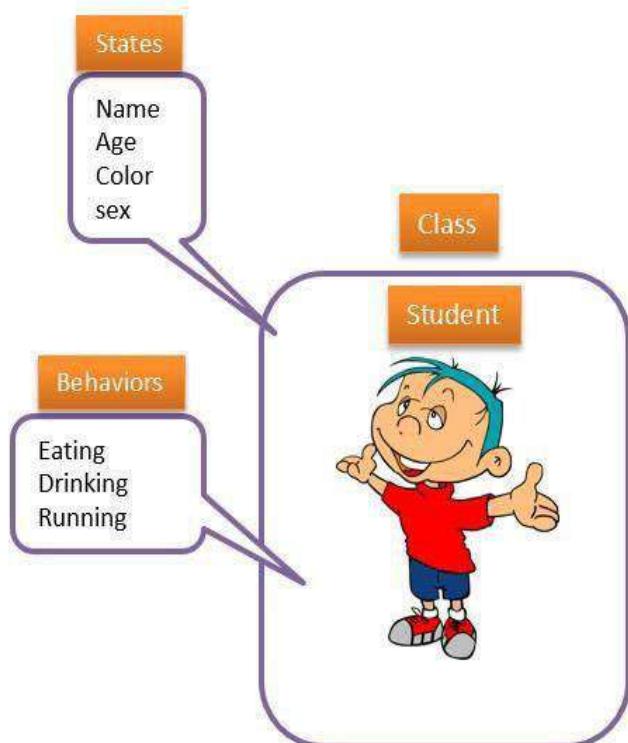
Breed = Chow Chow
Size = Midium
Age = 3 years
Color = Brown



A Student Class



What Is an Object?



- ✓ Object are same as real world objects.
- ✓ Java objects has both states and behaviors.



Creating A Class



- Defining a class is simple in **Python**.
- We start with the **class keyword** to indicate that we are creating a class, then we add the name of the class followed by a **colon**
- We can then add **class members**, which are **methods** and **attributes**



Syntax Of Creating A Class

Syntax:

```
class <class_name>:
```

```
    # class members
```

Example:

```
class Emp:
```

```
    pass
```

These Notes Have Python_world_In Notes Copyrights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Creating Objects

- In order to **use** a **class** we have to create it's object which is also called **instantiating** a class because **objects** are also called **instance** of the class
- So, to create an **instance** of a class, we use the **class name**, followed by **parentheses** and assign it to a **variable**.



Syntax Of Creating Object

Syntax:

var_name=class_name()

Example:

e=Emp()



Full Code

```
class Emp:  
    pass
```

```
e=Emp()  
print(type(e))  
print(e)
```

Output:

```
<class '__main__.Emp'>  
<__main__.Emp object at 0x0000000002cc8860>
```

1. The first line shows the **class name** which is **Emp**.

2. The second line shows the **address of the object** to which the **reference e is pointing**

3. The name **__main__** is the **name of the module** which Python automatically allots to our file

Adding Data Members/Attributes



- Once we have defined the class , our next step is to provide it data members/variables which can be used to hold values related to objects.
- In **Python** , a class can have **3 types** of variables:
 - Instance Variables:** Created per instance basis
 - Local Variables:** Created locally inside a method and destroyed when the method execution is over
 - Class Variables:** Created inside a class and shared by every object of that class. Sometimes also called as **static variables**



What Is An Instance Variable?



- **Object variables** or **Instance Variables** are created by **Python** for **each individual object** of the class.
- In this case, ***each object has its own copy of the instance variable*** and they are not shared or related in any way to the field by the same name in a different object



Creating Instance Variables



- Creation of **instance variables** in **Python** is **entirely different** than **C++ or Java**
- In these languages , we declare the **data members** inside the class and when we **instantiate** the class , these members are **allocated space** .

Creating Instance Variables In C++



- For example in **C++**, we would write :

```
class Emp
{
    int age;
    char name[20];
    double salary;
    .....
    .....
};
```

These are
called
instance
variables in
C++

Now to use this **Emp** class we would say:

```
Emp obj;
```

Doing this will create an **object** in memory by the name **e** and will contain three **instance members** called as **age**, **name** and **salary**. Also this line will **automatically call** a special method called **constructor** for **initializing the object**.

Creating Instance Variables In Java



- In **Java**, we would write :

```
class Emp
{
    int age;
    String name;
    double salary;
    .....
    .....
}
```

These are
called
instance
variables in
Java

Now to use this **Emp** class we would say:

```
Emp obj=new Emp();
```

Doing this will create an **object** in **heap** with the **data members** as **age**, **name** and **salary** and the **reference e** will be pointing to that **object**. Here also the special method called **constructor** will be called **automatically** for **initializing the object**

Creating Instance Variables In Python



- But in Python we use a very special method called `__init__()`, to **create** as well as **initialize** an object's initial attributes by giving them their **default value**.
- Python calls this method **automatically**, as soon as the object of the class gets created.
- Since it is called **automatically**, we can say it is like a **constructor** in **C++ or Java**.



Full Code

```
class Emp:  
    def __init__(self):  
        print("Object created...")
```

```
e=Emp()
```

Output:

```
object created. . .
```

As you can observe ,
Python has
automatically called the
special method
__init__() as soon as
we have created the
object of Emp class



Another Example

```
class Emp:  
    def __init__(self):  
        print("Object created...")
```

```
e=Emp()  
f=Emp()  
g=Emp()
```

Output:

```
object created.  
object created.  
object created.
```



The argument **self** ?



- You must have noticed that the code is using an argument called **self** in the argument list of **__init__()**
- So , now **2 questions** arise , which are :
 - What is **self** ?
 - Why it is required ?



What Is **self** ?



- In **Python** , whenever we create an object , **Python** calls the method **__init__O**
- But while calling this method , **Python** also passes the **address of the object** , for which it is calling **__init__O** , as the **first argument**.
- Thus , when we define the **__init__O** method we must provide it **atleast one formal argument** which will receive the object's address .
- This argument is named as **self**



What If We Don't Create **self** ?

```
class Emp:  
    def __init__():  
        print("Object created...")
```

```
e=Emp()
```

As you can observe,
Python has generated an
exception , since it has
passed the object address
as argument while calling
the method `__init__()` but
we have not declared any
argument to receive it

Output:

```
e=Emp()  
TypeError: __init__() takes 0 positional arguments but 1 was given
```

Can We Give Some Other Name To **self** ?



```
class Emp:  
    def __init__(myself):  
        print("Object created...")
```

```
e=Emp()
```

Output:

```
Object created. . .
```

As you can observe,
Python has allowed us to
use the name **myself**
instead of **self** , but the
convention is to always use
the word **self**



More About **self**



- Python always passes **address of the object** to every **instance method** of our class whenever we call it, not only to the method `__init__()`
- So, every **instance method** which we define in our class has to compulsorily have atleast one argument of type **self**



More About **self**



- The argument **self** always **points** to the **address of the current object**
- We can think it to be like **this reference** or **this pointer** of **Java** or **C++** languages



Is **self** A Keyword ?



- **No , not at all**
- Many programmers wrongly think **self** to be a **keyword** but it is not so.
- It is just a name and can be changed to anything else but the convention is to always use the name **self**
- **Another Important Point!**
- The argument **self** is **local** to the method body , so we cannot use it outside the method



Guess The Output

```
class Emp:  
    def __init__(self):  
        print("Object Created...")
```

```
e=Emp()  
print(self)
```

Output:

```
print(self)  
NameError: name 'self' is not defined
```

The Most Important Role Of **self**



- We can also use **self** to **dynamically** add **instance members** to the **current object**.
- To do this ,we simply have to use **self** followed by **dot operator** followed by **name** of the variable along with it's **initial value**
- **Syntax:**

```
class <class_name>:  
    def __init__(self):  
        self.<var_name>=value
```

-
-



Example

```
class Emp:  
    def __init__(self):  
        self.age=25  
        self.name="Rahul"  
        self.salary=30000.0
```

```
e=Emp()
```

```
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.
```

The variables **self.age**, **self.name** and **self.salary** are called **instance variables**

Remember , we cannot use **self** outside the class . So outside the class we will have to use the **reference variable e**

Another very important point to understand if you are from C++ background is that in Python by default everything in a class is public . So we can directly access it outside the class.



A Very Important Point!



- The **instance variables** called **age** , **name** and **salary** are accessed in **2 ways** in **Python**:
 - Inside the methods of the class , they are always accessed using **self** so that **Python** will refer them for **current object**
 - Outside the class , we cannot access them using **self** because **self is only available within the class.**
 - So outside the class we have to access them using the **object reference** we have created



Guess The Output ?

```
class Emp:
```

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

```
        self.age=25
```

```
        self.name="Rahul"
```

```
e=Emp()
```

```
e.salary=30000.0
```

```
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
```

Unlike C++ or Java ,
in Python we can
create instance
variables outside the
class by directly using
the object reference

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
```



A Problem With The Code



- Although the code works fine , but it has one problem .
- The problem is that for **every object** of Emp class , **Python** will call `__init__()` method and thus every object will be **initialized** with the **same values**
- To overcome this problem we can make the method `__init__()` parameterized

Passing Parameters To `__init__()`



- Since `__init__()` is also a method so just like other methods we can pass **arguments** to it .
- But we need to remember 2 things for this:
 - Since `__init__()` is called by **Python** at the time of **object creation** so we will have to pass these arguments at the time of **creation of the object**
 - We will have to define **parameters** also while defining `__init__()` to receive these **arguments**
- Finally using these **parameters** we can **initialize** instance members to **different values** for **different objects**

Passing Parameters

To __init__O



```
class Emp:
```

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

The variables age, name and salary are called local variables

```
e=Emp(25,"Rahul",30000.0)
```

```
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
```

```
f=Emp(31,"Varun",45000.0)
```

```
print("Age:",f.age,"Name:",f.name,"Salary:",f.salary)
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
```

```
Age: 31 Name: Varun Salary: 45000.0
```



An Important Point



- The argument **self**, should always be the first argument as **Python** passes the address of the current object as the first argument
- The **variables age , name and salary** used in the argument list of `__init__()` are called **parameters** or **local variables**.
- They will only **survive** until the method is **under execution** and after that they will be **destroyed by Python**



An Important Point



- Any **variable** declared inside the body of any method inside the class without using **self** will also be called as **local variable**
- It is a **common convention** to give **parameters** the **same name** as **instance members**, but it is not at all compulsory.

Passing Parameters To __init__O



```
class Emp:  
    def __init__(self,x,y,z):  
        self.age=x  
        self.name=y  
        self.salary=z
```

```
e=Emp(25,"Rahul",30000.0)  
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)  
f=Emp(31,"Varun",45000.0)  
print("Age:",f.age,"Name:",f.name,"Salary:",f.salary)
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0  
Age: 31 Name: Varun Salary: 45000.0
```



Guess The Output ?

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

```
e1=Emp("amit")  
e2=Emp("sumit",23)  
e3=Emp("deepak",34,50000.)  
print(e1.name)  
print(e2.name,e2.age)  
print(e3.name,e3.age,e3.sal)
```

Output:

```
e1=Emp("amit")  
TypeError: __init__() missing 2 required positional arguments: 'age' and 'sal'
```



Why Didn't The Code Run ?



- Recall , that we have already discussed that **PYTHON DOES NOT SUPPORT METHOD/FUNCTION OVERLOADING .**
- So if **two methods** have **same name** then the **second copy** of the method will **overwrite** the **first copy**.
- So , in our case **Python** remembers only one **__init__()** method , which is defined last and since it is taking **3 arguments** (excluding self) so our call:

e1=Emp("amit")

generated the exception



Question ?



- Can we do something so that the code runs with different number of arguments passed to Emp objects ?
- Yes !
- The solution is to use **default arguments**



Solution



```
class Emp:  
    def __init__(self,name,age=0,sal=0.0):  
        self.name=name  
        self.age=age  
        self.sal=sal
```

```
e1=Emp("amit")  
e2=Emp("sumit",23)  
e3=Emp("deepak",34,50000.)  
print(e1.name)  
print(e2.name,e2.age)  
print(e3.name,e3.age,e3.sal)
```

Output:

```
amit  
sumit 23  
deepak 34 50000 . 0
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 39



Today's Agenda

• **Introduction To Object Oriented Programming-II**

- Types Of Methods
- Adding Instance Methods
- Obtaining Details Of Instance Variables
- Different Ways To Create Instance Variables
- Deleting Instance Variables

Adding Methods In Class



- Once we have defined the class , our next step is to provide methods in it
- In **Python** , a class can have **3 types** of methods:
 - Instance Methods:** Called using object
 - Class Methods:** Called using class name
 - Static Methods:** Called using class name



Adding Instance Methods



- **Instance methods** are the most common type of methods in Python classes.
- These are called **instance methods** because they can access **instance members** of the object.



Adding Instance Methods



- These methods always take **atleast one parameter**, which is normally called **self**, which points to the **current object** for which the method is called.
- Through the **self** parameter, **instance methods** can access **data members** and other methods on the same object.
- This gives them a lot of power when it comes to **modifying** an **object's state**.



Example

```
class Emp:  
    def __init__(self,age,name,salary):  
        self.age=age  
        self.name=name  
        self.salary=salary  
    def show(self):  
        print("Age:",self.age,"Name:",self.name,"Salary:",self.salary)  
  
e=Emp(25,"Rahul",30000.0)  
f=Emp(31,"Varun",45000.0)  
e.show()  
f.show()
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0  
Age: 31 Name: Varun Salary: 45000.0
```

Exercise



- Write a program to create a class called **Circle** , having an instance member called **radius**. Provide following instance methods in your class:
 - **__init__()**: This method should initialize radius with the parameter passed
 - **cal_area()**: This method should calculate and print the area of the Circle
 - **cal_circumference()**: This method should calculate and print the circumference of the Circle
- Finally , in the main script , create a **Circle** object , **initialize radius** with **user input** and calculate and display it's **area** and **circumference**

Output:

```
Enter radius:10
Area of circle is 314.1592653589793
Circumference of circle is 62.83185307179586
```



Solution



```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def cal_area(self):
        area = math.pi * math.pow(self.radius, 2)
        print("Area of circle is", area)
    def cal_circumference(self):
        circumf = math.tau * self.radius
        print("Circumference of circle is", circumf)

radius = int(input("Enter radius:"))
cobj = Circle(radius)
cobj.cal_area()
cobj.cal_circumference()
```



Guess The Output ?

```
class Emp:  
  
    def __init__(self):  
        self.name="Amit"  
        self.age=24  
        self.sal=50000.0  
  
    def show(self):  
        print(age,name,sal)  
  
e1=Emp()  
e1.show()
```

Why did the code give exception?

The syntax we are using for accessing **name** , **age** and **sal** is only applicable to **local variables** and not for **instance members**.

And since there are no **local variables** by the name of **name** , **age** and **sal** , so the code is giving exception

Output:

```
print(age,name,sal)
NameError: name 'age' is not defined
```



Guess The Output ?

```
class Emp:  
  
    def __init__(self,name,age,sal):  
        self.name=name  
        self.age=age  
        self.sal=sal  
  
    def show(self):  
        print(age,name,sal)  
  
e1=Emp("amit",34,50000.0)  
e1.show()
```

Why did the code give exception?

The variables **name**, **age** and **sal** are **local variables** declared inside the method **__init__()** and hence are not available to the method **show()** , so the code gave **NameError** exception

Output:

```
print(age,name,sal)
NameError: name 'age' is not defined
```



Guess The Output ?



class Emp:

```
def __init__(self):
    self.name="Amit"
    self.age=24
    self.sal=50000.0
```

```
def show(self):
    print(self.age,self.name,self.sal)
```

e1=Emp()

e1.show()

Output:

24 Amit 50000.0

Obtaining Details Of Instance Variables



- Every object in Python has an **attribute** denoted by **__dict__**.
- This **attribute** is automatically added by Python and it contains all the **attributes** defined *for the object itself*.
- It maps the **attribute name** to its **value**.



Guess The Output ?



class Emp:

```
def __init__(self):
    self.name="Amit"
    self.age=24
    self.sal=50000.0
```

```
e1=Emp()
print(e1.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
```



Guess The Output ?



class Emp:

```
def __init__(self):
    self.name="Amit"
    self.age=24
    sal=50000.0
```

e1=Emp()

print(e1.__dict__)

Output:

```
{'name': 'Amit', 'age': 24}
```



Guess The Output ?



class Emp:

```
def __init__(self):
    self.name="Amit"
    self.age=24
def set_sal(self):
    self.sal=50000.0
```

```
e1=Emp()
print(e1.
      __dict__)
e1.set_sal()
print(e1.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24}
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
```



Guess The Output ?

```
class Emp:  
    def __init__(self):  
        self.name="Amit"  
        self.age=24  
        self.sal=50000.0  
  
    def show(self):  
        print(self.name,self.age,self.sal,self.department)  
  
e1=Emp()  
print(e1.__dict__)  
e1.__dict__['department']='IT'  
print(e1.__dict__)  
e1.show()
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}  
{'name': 'Amit', 'age': 24, 'sal': 50000.0, 'department': 'IT'}  
Amit 24 50000.0 IT
```

Since __dict__ is a dictionary , we can manipulate it and add/del instance members from it

How Many Ways Are There To Create Instance Variables ?



- Till now we can say there are **4 ways** in **Python** to create **instance variables**:
 - Inside the **constructor**/`__init__()` method using **self**
 - Inside **any instance method** of the class using **self**
 - **Outside the class** using it's **object reference**
 - Using the instance attribute `__dict__`



Guess The Output ?

```
class Emp:  
    def __init__(self,name,age,sal):  
        self.name=name  
        self.age=age  
        self.sal=sal  
  
    def setDept(self,department):  
        self.department=department  
  
    def setProject(self,project):  
        self.project=project  
  
    def setBonus(self,bonus):  
        self.bonus=bonus
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 30000.0, 'department': 'Finance', 'project': 'Banking Info System', 'bonus': 20000.0}  
  
{'name': 'Sumit', 'age': 34, 'sal': 45000.0, 'department': 'Production'}
```

```
e1=Emp("Amit",24,30000.0)  
e2=Emp("Sumit",34,45000.0)  
e1.setDept("Finance")  
e1.setProject("Banking Info System")  
e1.setBonus(20000.0)  
e2.setDept("Production")  
print(e1.__dict__)  
print()  
print(e2.__dict__)
```

Since Python is dynamically typed language so object's of same class can have different number of instance variables



Deleting Instance Variables



- We can **delete/remove** instance variables in 2 ways:
 - Using `del self.<var_name>` from the body of any **instance method** within the class
 - Using `del <obj_ref>.<var_name>` from **outside the class**



Guess The Output ?

```
class Boy:  
    def __init__(self,name,girlfriend):  
        self.name=name  
        self.girlfriend=girlfriend  
    def breakup(self):  
        del self.girlfriend  
  
b1=Boy("Deepak","Jyoti")  
print(b1.__dict__)  
b1.breakup()  
print(b1.girlfriend)
```

Output:

```
{'name': 'Deepak', 'girlfriend': 'Jyoti'}  
Traceback (most recent call last):  
  File "classdemo7.py", line 10, in <module>  
    print(b1.girlfriend)  
AttributeError: 'Boy' object has no attribute 'girlfriend'
```



Guess The Output ?

```
class Engineer:  
    def __init__(self, girlfriend, job):  
        self.girlfriend = girlfriend  
        self.job = job
```

```
    def fired(self):  
        del self.job
```

```
e1 = Engineer("Rani", "Software Engineer")
```

```
print(e1.__dict__)
```

```
e1.fired()
```

```
del e1.girlfriend
```

```
print(e1.__dict__)
```

Output:

```
{'girlfriend': 'Rani', 'job': 'Software Engineer'}  
{}
```



Guess The Output ?

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

```
e1=Emp("Amit",24,50000.0)
```

```
print(e1.__dict__)
```

```
del e1
```

```
print(e1.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}  
Traceback (most recent call last):  
  File "classdemo8.py", line 11, in <module>  
    print(e1.__dict__)  
NameError: name 'e1' is not defined
```



Guess The Output ?

```
class Emp:  
    def __init__(self,name,age,sal):  
        self.name=name  
        self.age=age  
        self.sal=sal  
    def remove(self):  
        del self  
  
e1=Emp("Amit",24,50000.0)  
print(e1.__dict__)  
e1.remove()  
print(e1.__dict__)
```

Since the object pointed by **self** is also pointed by **e1** , so Python didn't remove the object , rather it only removes the reference **self**

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}  
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
```



Guess The Output ?

```
class Emp:  
    def __init__(self,name,age,sal):  
        self.name=name  
        self.age=age  
        self.sal=sal  
  
e1=Emp("Amit",24,50000.0)  
e2=Emp("Sumit",25,45000.0)  
print(e1.__dict__)  
print(e2.__dict__)  
del e1.sal  
del e2.age  
print()  
print(e1.__dict__)  
print(e2.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}  
{'name': 'Sumit', 'age': 25, 'sal': 45000.0}  
  
{'name': 'Amit', 'age': 24}  
{'name': 'Sumit', 'sal': 45000.0}
```

Since **instance variables** have a **separate copy** created for **every object**, so **deleting an instance variable** from **one object** will not effect the other object's same instance variable



PYTHON

LECTURE 40

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Today's Agenda

• **Introduction To Object Oriented Programming-III**

- Adding Class Variables
- Different Ways To Create A Class Variable
- Different Ways To Access A Class Variable
- Obtaining Details Of Class Variables
- Deleting Class Variables



Class Variables



- **Class variables** are those variables which are defined within the **class body outside any method**
- They are also called as **static variables** , although there is no **static** keyword used with them



Class Variables



- They are **shared by all instances** of the class and **have the same value** for each instance of the class.
- They have a **single copy** maintained at the **class level**



What Is **Class Level** ?



- The term **class level** means inside the **class object**.
- In **Python** , *for every class one special object is created called as **class object***
- **Don't think it is the same object which we create. No it is not that!**
- Rather , for every class , **Python** itself creates an object called as **class object** and inside this object all the **class / static** variables live

When Should We Use Class Variable ?



- Whenever we don't want to create a **separate copy** of the **variable** for **each object** , then we can declare it as a **class variable**.
- For example :
 - The variable **pi** in a class called **Circle** can be declared as a **class level variable** since all **Circle objects** will have the **same value** for **pi**
 - Another example could be a variable called **max_marks** in a class called **Student** . It should also be declared at the **class level** because each **Student** will have same **max_marks**



Using Class Variable



- We can use a class variable at **6 places** in **Python**:
 - Inside the **class body** but **outside any method**
 - Inside the **constructor** using the **name of the class**
 - Inside **instance method** using **name of the class**
 - Inside **classmethod** using **name of the class** or using the special reference **cls**
 - Inside **staticmethod** using the **name of the class**
 - From outside the class using **name of the class**



Declaring Inside Class Body



- To declare a **class variable** inside class body but outside any method body , we simply declare it below the **class header**

- Syntax:**

```
class <class_name>:  
    <var_name>=<value>  
    def __init__(self):  
        // object specific code
```

This is called a
class variable

- To access the **class level variables** we use **class name** before them with **dot operator**

How To Access and Modify Class Variables?



- We must clearly understand the difference between **accessing** and **modifying** .
- **Accessing** means we are just reading the value of the variable
- **Modifying** means we are changing it's value

How To Access Class Variables?



- The **class variables** can be **accessed** in **4** ways:
 - Using **name of the class** anywhere in the program
 - Using **self** inside any **instance method**
 - Using **object reference** outside the class
 - Using special reference **cls** inside **classmethod**

How To Modify Class Variables?



- The **class variables** can be **modified** in **3** ways:
 - Using **name of the class** anywhere inside the methods of the class
 - Using special reference **cls** inside **classmethod**
 - Using **name of the class** outside the class body
- **Special Note:** We must never **modify** a **class variable** using **self** or **object reference**, because it will not **modify** the **class variable**, rather will create a new **instance variable** by the same name



Example

```
class CompStudent:  
    stream = 'cse'  
    def __init__(self,name,roll):  
        self.name = name  
        self.roll = roll  
  
obj1 = CompStudent('Atul',1)  
obj2 = CompStudent('Chetan', 2)  
print(obj1.name)  
print(obj1.roll)  
print(obj1.stream)  
print(obj2.name)  
print(obj2.roll)  
print(obj2.stream)  
print(CompStudent.stream)
```

The variable **stream** is class variable

Everytime we will access the class variable **stream** from any object , the value will remain same

Output:

```
Atul  
1  
cse  
Chetan  
2  
cse  
cse
```

Exercise



- Write a program to create a class called **Emp** , having 3 **instance members** called **name** , **age** and **sal** . Also declare a **class variable** called **raise_amount** to store the **increment percentage** of **sal** and set it to **7.5** .
- Now provide following methods in your class
 - **__init__()** : This method should initialize instance members with the parameter passed
 - **increase_sal()**: This method should calculate the increment in sal and add it to the instance member sal
 - **display()**: This method should display name , age and sal of the employee
- Finally , in the main script , **create 2 Emp objects** , **initialize them** and **increase their salary** . Finally **display** the data

Output:

```
Before incrementing :
```

```
Amit 24 50000.0
Sumit 26 45000.0
```

```
After incrementing by 7.5 percent:
```

```
Amit 24 53750.0
Sumit 26 48375.0
```



Solution

```
class Emp:  
    raise_amount=7.5  
    def __init__(self,name,age,sal):  
        self.name=name  
        self.age=age  
        self.sal=sal  
    def increase_sal(self):  
        self.sal=self.sal+(self.sal*Emp.raise_amount/100)  
    def display(self):  
        print(self.name,self.age,self.sal)  
  
e1=Emp("Amit",24,50000.0)  
e2=Emp("Sumit",26,45000.0)  
print("Before incrementing :")  
print("_____");  
e1.display()  
e2.display()  
e1.increase_sal()  
e2.increase_sal()  
print()  
print("After incrementing by",Emp.raise_amount,"percent:")  
print("_____");  
e1.display()  
e2.display()
```

Declaring Class Variable Inside Constructor



- We can declare a **class variable** inside the **constructor** also by **prefixing** the variable name with the **name of the class** and **dot** operator

- **Syntax:**

```
class <class_name>:
```

```
def __init__(self):  
<class name>.<var_name>=<value>  
self.<var_name>=<value>
```

This is called a
class variable

-
-
-



Example

```
class CompStudent:
```

```
    def __init__(self,name,roll):
        CompStudent.stream='cse'
        self.name = name
        self.roll = roll
```

```
obj1 = CompStudent('Atul',1)
obj2 = CompStudent('Chetan', 2)
print(obj1.name)
print(obj1.roll)
print(obj1.stream)
print(obj2.name)
print(obj2.roll)
print(obj2.stream)
print(CompStudent.stream)
```

We have shifted the var decl from class body to constructor body , but still it will be treated as class variable because we have prefixed it with classnname

Output:

```
Atul
1
cse
Chetan
2
cse
cse
```

Declaring Class Variable Inside Instance Method



- We can declare a **class variable** inside an instance method also also by **prefixing** the variable name with the **name of the class** and **dot** operator

- Syntax:**

```
class <class_name>:
```

This is called a
class variable

```
def <method_name>(self):  
<class name>.<var_name>=<value>  
self.<var_name>=<value>
```

-
-
-



Example

```
class Circle:  
    def __init__(self, radius):  
        self.radius = radius  
    def cal_area(self):  
        Circle.pi = 3.14  
        self.area = Circle.pi * self.radius ** 2  
c1 = Circle(10)  
c2 = Circle(20)  
c1.cal_area()  
  
print("radius =", c1.radius, "area =", c1.area, "pi =", Circle.pi)  
c2.cal_area()  
  
print("radius =", c2.radius, "area =", c2.area, "pi =", Circle.pi)
```

Output:

```
radius = 10 area = 314.0 pi = 3.14  
radius = 20 area = 1256.0 pi = 3.14
```

We have shifted the var decl from class body to method body , but still it will be treated as class variable because we have prefixed it with classnname

Obtaining Details Of Class Variables



- As we know , **class variables** are owned by a class itself (i.e., by its definition), so to store their details a class also uses a dictionary called **__dict__**
- Thus we can see that Python has **2 dictionaries** called **__dict__**.
- One is ***<class_name>.__dict__*** and the other is ***<object_ref>.__dict__***



Guess The Output ?

```
class Emp:  
    raise_per=7.5  
    comp_name="Google"  
    def __init__(self):  
        self.name="Amit"  
        self.age=24  
        self.sal=50000.0  
  
e1=Emp()  
print(e1.__dict__)  
print()  
print(Emp.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
```

```
{'__module__': '__main__', 'raise_per': 7.5, 'comp_name': 'Google', '__init__':  
<function Emp.__init__ at 0x0000000028179D8>, '__dict__': <attribute '__dict__'  
of 'Emp' objects>, '__weakref__': <attribute '__weakref__' of 'Emp' objects>,  
__doc__: None}
```

How many class variables will be created by this code?



```
class Sample:  
    i=10  
    def __init__(self):  
        Sample.j=20  
    def f1(self):  
        Sample.k=30  
    Sample.m=40  
    print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0000000022A79D8>, 'f1': <function Sample.f1 at 0x000000000022A7A60>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None, 'm': 40}
```

Why the code is showing only **2 class variables** even though we have **4** ?

This is because the class variable **k** will only be created when **f1()** gets called . Similarly the variable **j** will be created when we will create any object of the class . But since we didn't create any object nor we have called the method **f1()** so only **2 class variables** are there called **i** and **m**

How many class variables will be created by this code?



```
class Sample:  
    i=10  
    def __init__(self):  
        Sample.j=20  
    def f1(self):  
        Sample.k=30  
    Sample.m=40  
s1=Sample()  
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x00000002DD79D8>, 'f1': <function Sample.f1 at 0x0000000002DD7A60>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None, 'm': 40, 'j': 20}
```

Three **class variables** will be created by the code called **i,j and m**

How many class variables will be created by this code?



```
class Sample:  
    i=10  
    def __init__(self):  
        Sample.j=20  
    def f1(self):  
        Sample.k=30  
  
    Sample.m=40  
    s1=Sample()  
    S2=Sample()  
    print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x000000002DD79D8>, 'f1': <function Sample.f1 at 0x000000002DD7A60>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None, 'm': 40, 'j': 20}
```

Still only three **class variables** will be created by the code called **i,j** and **m** because **class variables** are not created **per instance basis** rather there is only **1 copy** shared by all the objects

How many class variables will be created by this code?



```
class Sample:  
    i=10  
    def __init__(self):  
        Sample.j=20  
    def f1(self):  
        Sample.k=30  
  
Sample.m=40  
s1=Sample()  
s2=Sample()  
s1.f1()  
s2.f1()  
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0000000029779D8>, 'f1': <function Sample.f1 at 0x000000002977A60>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None, 'm': 40, 'j': 20, 'k': 30}
```



Guess The Output ?



```
class Sample:  
    i=10  
    def __init__(self):  
        print("Constructor called...")  
        print(Sample.i)  
        print(self.i)  
    def f1(self):  
        print("f1 called...")  
        print(Sample.i)  
        print(self.i)  
  
s1=Sample()  
s1.f1()
```

Output:

```
Constructor called...  
10  
10  
f1 called...  
10  
10
```



Guess The Output ?

```
class Sample:
```

```
i=10
```

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

```
    self.i=20
```

```
s1=Sample()
```

```
print(Sample.i)
```

Output:

```
10
```

As mentioned previously , if we use **self** or **object reference** to modify a **class variable** , then **Python** does not **modify** the **class variable** . Rather it creates a new **instance variable** inside the **object's memory area** by the same name.

So in our case **2 variables** by the name **i** are created . One as **class variable** and other as **instance variable**



Guess The Output ?

class Sample:

i=10

def __init__(self):

self.i=20

s1=Sample()

print(Sample.i)

print(s1.i)

Output:

**10
20**



Guess The Output ?



class Sample:

i=10

def __init__(self):

Sample.i=20

s1=Sample()

print(Sample.i)

print(s1.i)

Output:

**20
20**



Guess The Output ?



class Sample:

i=10

def __init__(self):

Sample.i=20

s1=Sample()

s1.i=30

print(Sample.i)

print(s1.i)

Output:

**20
30**



Guess The Output ?

class Sample:

i=10

def __init__(self):

self.j=20

s1=Sample()

s2=Sample()

s1.i=100

s1.j=200

print(s1.i,s1.j)

print(s2.i,s2.j)

Output:

**100 200
10 20**



Guess The Output ?

class Sample:

i=10

def f1(self):

self.j=20

s1=Sample()

s2=Sample()

s1.i=100

s1.j=200

print(s1.i,s1.j)

print(s2.i,s2.j)

Output:

100 200

Traceback (most recent call last):

File "classdemo15.py", line 11, in <module>
 print(s2.i,s2.j)

AttributeError: 'Sample' object has no attribute 'j'



Deleting Class Variables



- We can **delete/remove** instance variables in 2 ways
 - Using **del classname.<var_name>** from anywhere in the program
 - Using **del cls.<var_name>** from **classmethod**
- **Special Note:** We cannot **delete** a **class variable** using **object reference** or **self**, otherwise **Python** will throw **AttributeError** exception



Guess The Output ?

class Sample:

i=10

def __init__(self):

 del Sample.i

print(Sample.__dict__)

s1=Sample()

print()

print(Sample.__dict__)

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x00000002A379D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None}
```

```
{'__module__': '__main__', '__init__': <function Sample.__init__ at 0x00000000002A379D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None}
```



Guess The Output ?

class Sample:

i=10

def __init__(self):

 del self.i

print(Sample.__dict__)

s1=Sample()

print()

print(Sample.__dict__)

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x00000002B079D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None}
```

Traceback (most recent call last):

```
  File "classdemo15.py", line 7, in <module>
    s1=Sample()
```

```
  File "classdemo15.py", line 4, in __init__
    del self.i
```

```
AttributeError: i
```



Guess The Output ?

```
class Sample:  
    i=10  
    def __init__(self):  
        del Sample.i  
  
print(Sample.__dict__)  
s1=Sample()  
del Sample.i  
print()  
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x000000002DE79D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None}  
Traceback (most recent call last):  
  File "classdemo15.py", line 8, in <module>  
    del Sample.i  
AttributeError: i
```



PYTHON

LECTURE 41

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Today's Agenda

- **Introduction To Object Oriented Programming-IV**
 - Class Methods
 - Creating Class Methods
 - Accessing Class Methods
 - Static Methods
 - Accessing Static Methods
 - Difference Between Instance Method , Class Method and Static Methods



Class Methods



- Just like we can have **class variables** , similarly **Python** also allows us to create **class methods**.
- These are those methods ***which work on the class as a whole*** , instead of working on it's **object**.
- For , example in our **Emp** class if we want to initialize the class variable **raise_per** inside a method , then the best way would be to create a **class method** for this purpose



Creating A Class Method

- To create a **class method** we write the special word **@classmethod** on top of method definition

- Syntax:**

```
class <class_name>:  
    @classmethod  
        def <method_name>(cls):  
            // class specific code
```

This is called
decorator

Notice that a **class method**
gets a special **object reference** passed as
argument by Python
called as **class reference**

Important Points About ClassMethods



- To define a **class method** it is compulsory to use the decorator **@classmethod**
- **ClassMethods** can only access **class level data** and not **instance specific data**

Important Points About ClassMethods



- Just like **Python** passed **self** as argument to **instance methods**, it automatically passes **cls** as argument to **classmethods**
- The argument **cls** is always passed as the first argument and represents the **class object**.

Important Points About ClassMethods



- Recall , that for every class **Python** creates a special object called class object , so the reference **cls** points to this object.
- The name **cls** is just a convention , although we can give any name to it.

Important Points About ClassMethods



- To call a **classmethod** we simply prefix it with **classname** followed by dot operator.
- Although we can use **object reference** also to call a **classmethod** but *it is highly recommended not to do so* , since **classmethods** do not work upon **individual instances** of the class

Exercise



- Write a program to create a class called **Emp** , having an **instance members** called **name** , **age** and **sal** . Also declare a **class variable** called **raise_amount** to store the **increment percentage** of **sal** and **set it the value given by the user**
- Now provide following methods in your class
 - **__init__()**: This method should initialize instance members with the parameter passed
 - **increase_sal()**: This method should calculate the increment in sal and add ot to the instance member sal
 - **display()**: This method should display name , age and sal of the employee
- Finally , in the main script , **create 2 Emp objects** , **initialize them** and **increase their salary** . Finally **display** the data

Output:

```
Enter raise percentage:8.5
Before incrementing :
Amit 24 50000.0
Sumit 26 45000.0
After incrementing by 8.5 percent:
Amit 24 54250.0
Sumit 26 48825.0
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 42



Today's Agenda

- **Advance Concepts Of Object Oriented Programming-I**
 - Encapsulation
 - Does Python Support Encapsulation ?
 - How To Declare Private Members In Python ?
 - The object Class And The `__str__()` Method
 - The Destructor



Encapsulation



- **Encapsulation** is the packing of *data and functions operating on that data* into a **single component** and *restricting the access to some of the object's components*.
- **Encapsulation** means that the internal representation of an object is **generally hidden** from view **outside of the class body**.

Is The Following Code Supporting Encapsulation ?



```
class Emp:  
    def __init__(self):  
        self.age=25  
        self.name="Rahul"  
        self.salary=30000.0
```

```
e=Emp()  
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
```

No , the following code is violating **Encapsulation** as it is allowing us to **access data members** from **outside the class** directly using object

How To Achieve Encapsulation In Python ?



- To achieve **Encapsulation** in **Python** we have to prefix the data member name with **double underscore**
- **Syntax:**

self.__<var_name>=<value>



Achieving Encapsulation

```
class Emp:  
    def __init__(self):  
        self.age=25  
        self.name="Rahul"  
        self.__salary=30000.0
```

```
e=Emp()  
print("Age:",e.age)  
print("Name:",e.name)  
print("Salary:",e.__salary)
```

Output:

```
Age: 25  
Name: Rahul  
Traceback (most recent call last):  
  File "classdemo22.py", line 10, in <module>  
    print("Salary:",e.__salary)  
AttributeError: 'Emp' object has no attribute '__salary'
```

Since we have created the data member as `__salary` so it has become a **private member** and cannot be accessed outside the class directly



Achieving Encapsulation



- Now to access such **private members**, we must define **instance methods** in the class
- From **outside the class** we must call these **methods** using **object** instead of directly accessing **data members**



Achieving Encapsulation

```
class Emp:  
    def __init__(self):  
        self.__age=25  
        self.__name="Rahul"  
        self.__salary=30000.0  
  
    def show(self):  
        print("Age:",self.__age,"Name:",self.__name,"Salary:",self.__salary)
```

```
e=Emp()  
e.show()
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
```



Private Methods



- Just like we have **private data members** , we also can have **private methods** .
- The syntax is also same.
- Simply **prefix the method name with double underscore** to make it a **private method**



Private Methods

```
class Emp:  
    def __init__(self):  
        self.__age=25  
        self.__name="Rahul"  
        self.__salary=30000.0  
    def __show(self):  
        print("Age:",self.__age,"Name:",self.__name,"Salary:",self.__salary)
```

```
e=Emp()  
e.__show()
```

Output:

```
Traceback (most recent call last):  
  File "classdemo22.py", line 10, in <module>  
    e.__show()  
AttributeError: 'Emp' object has no attribute '__show'
```



An Important Point



- When we declare a data member with double underscore indicating that it is private , **Python** actually masks it
- In other words , **Python** changes the name of the variable by using the syntax **_<classname>_<attributename>**
- **For example** , **__age** will actually become **_Emp__age**



So, What It Means To Us ?



- This means that **private attributes** are **not actually private** and are not prevented by **Python** from getting accessed from outside the class.
- So if they are **accessed** using the **above mentioned syntax** then no **Error** or **Exception** will arise
- So , finally we can say **NOTHING IN PYTHON IS ACTUALLY PRIVATE**



Accessing Private Data

```
class Emp:  
    def __init__(self):  
        self.__age=25  
        self.__name="Rahul"  
        self.__salary=30000.0  
  
    def show(self):  
        print("Age:",self.__age,"Name:",self.__name,"Salary:",self.__salary)  
  
e=Emp()  
e.show()  
print("Age:",e.__age,"Name:",e.__name,"Salary:",e.__salary)
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0  
Age: 25 Name: Rahul Salary: 30000.0
```



The `__str__()` Method



- In **Python**, whenever we try to print an **object reference** by passing it's name to the **print()** function , we get **2 types of outputs**:
 - For **predefined classes** like **list** ,**tuple** or **str** , we get the **contents of the object**
 - For **our own class objects** we get the **class name** and the **id** of the **object instance** (which is the object's memory address in **CPython.**)



Why Is It So ?



- This is because **whenever we pass an object reference name to the print() function , Python internally calls a special instance method** available in **our class**.
- This method is called **__str__()** .

From where this method came ?



- From **Python 3.0** onwards , every class which we create always automatically inherits from the class **object**
- Or , we can say that **Python** implicitly inherits our class from the class **object**.
- The class **object** defines some special methods which every class inherits .
- Amongst these special methods some very important are **__init__()**, **__str__()**, **__new__()** etc

Can we see all the members of object class ?



- Yes , it is very simple!
- Just create an instance of **object** class and call the function **dir()** .
- Recall that we used **dir()** to print names of all the **members** of a **module** .
- Similarly we also can use **dir()** to **print names** of all the members of any class by passing it the **instance** of the class as **argument**



Example

```
obj=object()
print(type(obj))
print(dir(obj))
```

Output:

```
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```



The `__str__()` Method



- Now , if we do not redefine (override) this method in our class , then **Python** calls it's **default implementation** given by **object class** which is designed in such a way that it **returns the class name** followed by **object's memory address**
- However all built in classes like **list , str , tuple , int , float , bool** etc have **overridden** this method in such a way that it returns the content of the object.



Overriding `__str__()`



- So if we also want the same behaviour for our object then we also can **override** this method in our class in such a way that it returns the **content of the object**.
- The only point we have to remember while **overriding** this method is that ***it should return a string value***



Overriding `__str__()`

```
class Emp:  
    def __init__(self,age,name,salary):  
        self.age=age  
        self.name=name  
        self.salary=salary  
    def __str__(self):  
        return f'Age:{self.age},Name:{self.name},Salary:{self.salary}'  
  
e=Emp(25,"Rahul",30000.0)  
print(e)
```

Output:

Age:25,Name:Rahul,Salary:30000.0



Destructor



- Just like a **constructor** is used to **initialize** an object, a **destructor** is used to destroy the object and perform the final clean up.
- But a question arises that if we already have **garbage collector** in **Python** to clean up the memory , then ***why we need a destructor ?***



Destructor



- Although in python we do have **garbage collector** to **clean up the memory**, but it's not just memory which has to be freed when an object is dereferenced or destroyed.
- There can be a **lot of other resources as well**, like **closing open files, closing database connections** etc.
- Hence when we might require a **destructor** in our class for this purpose



Destructor In Python



- Just like we have `__init__()` which can be considered like a constructor as it initializes the object , similarly in **Python** we have another magic method called `__del__()`.
- This method is automatically called by **Python** whenever an **object reference** goes **out of scope** and the **object** is **destroyed**.



Guess The Output ?

```
class Test:  
    def __init__(self):  
        print("Object created")  
  
    def __del__(self):  
        print("Object destroyed")
```

t=Test()

Output:

```
object created  
object destroyed
```

Since at the end of the code ,
Python collects the object
through it's garbage
collector so it automatically
calls the `__del__()` method
also

How To Force Python To Call __del__O ?



- If we want to force **Python** to call the __del__O method , then we will have to forcibly destroy the object
- To do this we have to use **del operator** passing it the **object reference**



Guess The Output ?



```
class Test:  
    def __init__(self):  
        print("Object created")  
  
    def __del__(self):  
        print("Object destroyed")  
t1=Test()  
del t1  
print("done")
```

Output:

```
Object created  
Object destroyed  
done
```



Guess The Output ?

```
class Test:  
    def __init__(self):  
        print("Object created")  
  
    def __del__(self):  
        print("Object destroyed")  
  
t1=Test()  
t2=t1  
  
del t1  
print("done")
```

Output:

```
Object created  
done  
Object destroyed
```

We must remember that Python destroys the object only when the reference count becomes 0 . Now in this case after deleting **t1** , still the object is being referred by **t2** . So the **__del__()** was not called on **del t1**. It only gets called when **t2** also goes out of scope at the end of the program and reference count of the object becomes 0



Guess The Output ?

```
class Test:  
    def __init__(self):  
        print("Object created")  
  
    def __del__(self):  
        print("Object destroyed")  
  
t1=Test()  
t2=t1  
del t1  
print("t1 deleted")  
del t2  
print("t2 deleted")  
print("done")
```

Output:

```
object created  
t1 deleted  
object destroyed  
t2 deleted  
done
```

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 43



Today's Agenda

• **Advance Concepts Of Object Oriented Programming-II**

- Inheritance
- Types Of Inheritance
- Single Inheritance
- Using super()
- Method Overriding

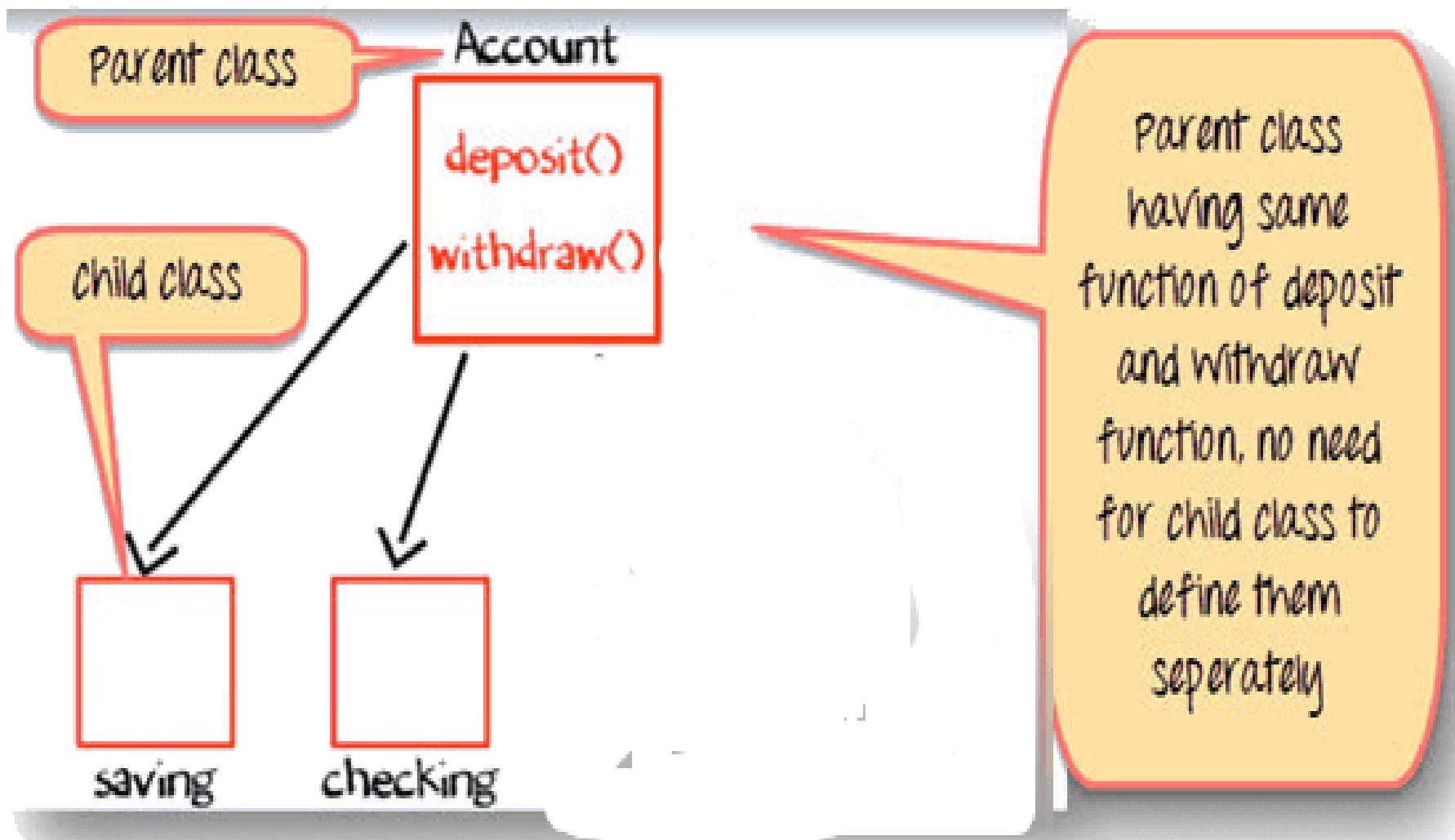


Inheritance



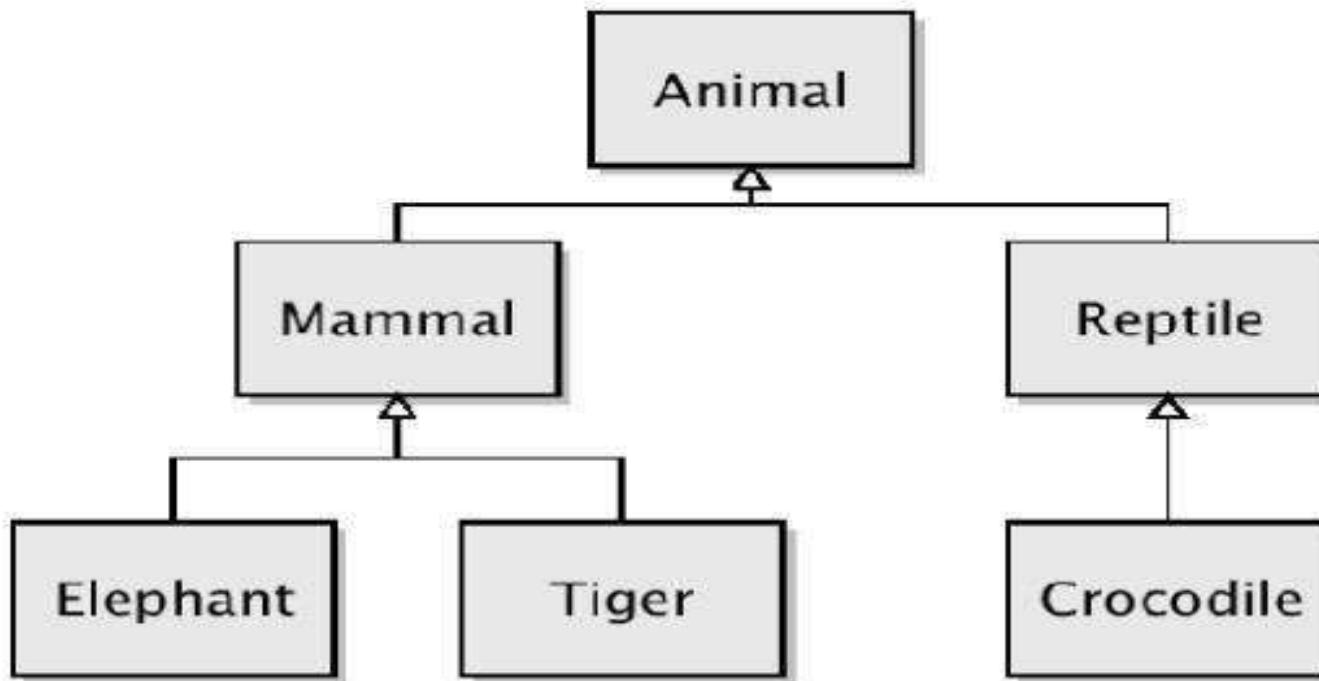
- Inheritance is a **powerful feature** in **object oriented programming**.
- It refers to defining a **new class** with **little or no modification** to an **existing class**.
- The **new class** is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Real Life Examples





Real Life Examples





Benefits

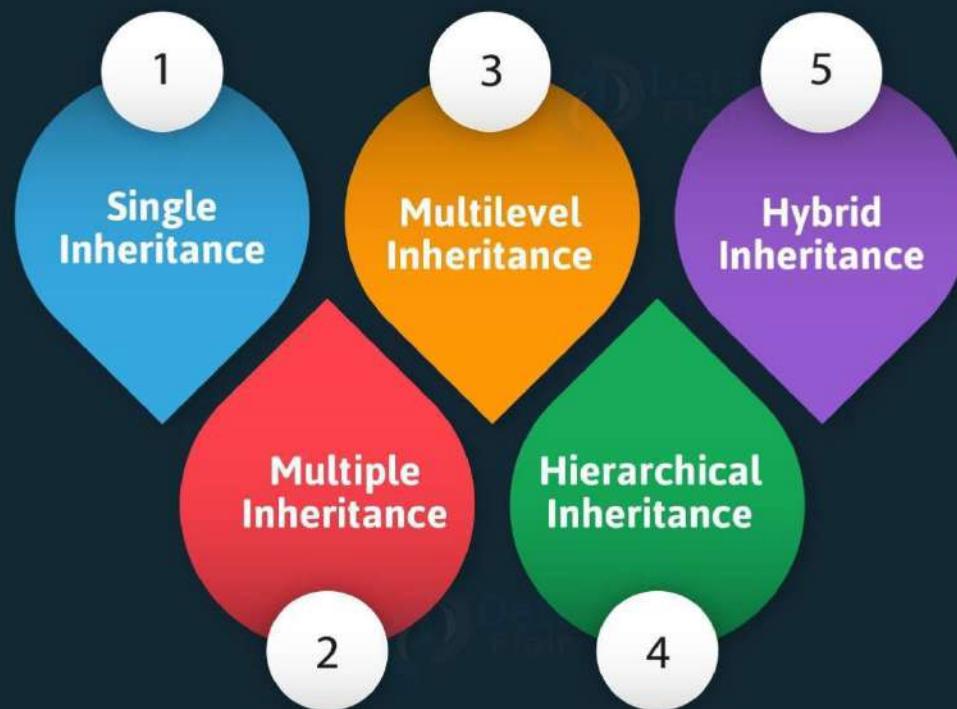


- It represents **real-world relationships** well.
- It provides **reusability** of a code. We **don't have to** write the **same code again and again**.
- It also allows us to **add more features** to a class without modifying it.

Types Of Inheritance Supported By Python



Types of Inheritance



Syntax Of Single Inheritance In Python



class BaseClass:
Body of base class

class DerivedClass(BaseClass):
Body of derived class

For Ex:

class Account:

pass

class SavingAccount(Account):
pass



Example

```
class Animal:  
    def eat(self):  
        print("It eats.")  
    def sleep(self):  
        print("It sleeps.")  
  
class Bird(Animal):  
    def set_type(self,type):  
        self.type=type  
    def fly(self):  
        print("It flies in the sky.")  
    def __str__(self):  
        return "This is a "+self.type;
```

```
duck=Bird()  
duck.set_type("Duck")  
print(duck)  
duck.eat()  
duck.sleep()  
duck.fly()
```

Output:

```
This is a Duck  
It eats.  
It sleeps.  
It flies in the sky.
```



Using super()

- In Python , to call parent class members from the child class we can use the method **super()**.
- Using **super()** is required in 2 situations:
 - For calling parent class constructor
 - For calling overridden methods

How Constructors Behave In Inheritance ?



- Whenever we create a **child class object** , **Python** looks for **__init__O** method in **child class**.
- If the **child class** doesn't contain an **_ init__O** method then **Python** goes up in the inheritance chain and looks for the **__init__O** method of **parent class**

How Constructors Behave In Inheritance ?



- If the parent class contains `__init__()` , then it executes it
 -
- Now an important point to notice is that if child class also has `__init__()` , then **Python** will not call parent's `__init__()` method.
- That is , unlike **Java** or **C++** , **Python** does not automatically call the parent class `__init__()` if it finds an `__init__()` method in **child class**

How Constructors Behave In Inheritance ?



```
class A:  
    def __init__(self):  
        print("Instantiating A...")
```

```
class B(A):  
    pass
```

```
b=B()
```

Output:

```
Instantiating A...
```

As you can see,
Python called the
constructor of class
A , since B class
doesn't have any
constructor defined

How Constructors Behave In Inheritance ?



```
class A:  
    def __init__(self):  
        print("Instantiating A...")  
  
class B(A):  
    def __init__(self):  
        print("Instantiating B...")
```

b=B()

Output:

Instantiating B...

This time , Python
did not call the
constructor of class
A as it found a
constructor in B
itself

How Constructors Behave In Inheritance ?



- So , what is the problem if parent constructor doesn't get called ?
- The problem is that , if parent class **constructor doesn't get** called then all the **instance members it creates** will **not be made available to child class**

How Constructors Behave In Inheritance ?



```
class Rectangle:
```

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

```
        self.l=10
```

```
        self.b=20
```

```
class Cuboid(Rectangle):
```

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

```
        self.h=30
```

```
    def volume(self):
```

```
        print("Vol of cuboid is",self.l*self.b*self.h)
```

```
obj=Cuboid()
```

```
obj.volume()
```

Output:

```
Traceback (most recent call last):
  File "inhdemo2.py", line 15, in <module>
    obj.volume()
  File "inhdemo2.py", line 10, in volume
    print("Vol of cuboid is",self.l*self.b*self.h)
AttributeError: 'Cuboid' object has no attribute 'l'
```

Since , constructor of **Rectangle** was not called , so the expression **self.l** produced exception because there is no **attribute** created by the name of **l**

How Can We Explicitly Call `__init__()` Of Parent Class ?



- If we want to call the parent class `__init__()`, then we will have 2 options:
 - Call it using the name of parent class explicitly
 - Call it using the method `super()`

Calling Parent Constructor Using Name



```
class Rectangle:  
    def __init__(self):  
        self.l=10  
        self.b=20  
  
class Cuboid(Rectangle):  
    def __init__(self):  
        Rectangle.__init__(self)  
        self.h=30  
    def volume(self):  
        print("Vol of cuboid is",self.l*self.b*self.h)
```

obj=Cuboid()
obj.volume()
Output:
Vol of cuboid is 6000

Notice that we have to explicitly pass the argument `self` while calling `__init__()` method of parent class

Calling Parent Constructor Using **super()**



```
class Rectangle:  
    def __init__(self):  
        self.l=10  
        self.b=20
```

```
class Cuboid(Rectangle):  
    def __init__(self):  
        super().__init__();  
        self.h=30  
    def volume(self):  
        print("Vol of cuboid is",self.l*self.b*self.h)
```

obj=Cuboid()
obj.volume()

Output:

Vol of cuboid is 6000

Again notice that this time we don't have to pass the argument **self** when we are using **super()** as Python will automatically pass it



What Really Is **super()** ?



- The method **super()** is a **special method** made available by **Python** which returns a **proxy object** that *delegates method calls to a parent class*
- In simple words the method **super()** provides us a special object that can be used to transfer call to parent class methods



Benefits Of `super()`



- A common question that arises in our mind is that why to use **`super()`**, if we can call the parent class methods using **parent class name**.
- The answer is that **`super()`** gives **4 benefits**:
 - We don't have to pass **`self`** while calling any method using **`super()`**.
 - If the **name of parent class changes** after inheritance then we will not have to rewrite the code in child **as `super()` will automatically connect itself to current parent**
 - It can be used to resolve **method overriding**
 - It is very helpful in **multiple inheritance**



Method Overriding

- To understand **Method Overriding**, try to figure out the output of the code given in the next slide

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Guess The Output ?

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

```
class Emp(Person):  
    def __init__(self,age,name,id,sal):  
        super().__init__(age,name)  
        self.id=id  
        self.sal=sal
```

```
e=Emp(24,"Nitin",101,45000)  
print(e)
```

Output:

Age : 24 , Name : Nitin



Explanation



- As we know , whenever we **pass** the name of an **object reference** as **argument** to the function **print()** , **Python** calls the method **__str__()**.
- But since the class **Emp** doesn't has this method , so **Python moves up in the inheritance chain** to find this method in the base class **Person**
- Now since the class **Person** has this method , **Python** calls the **__str__()** method of **Person** which returns only the **name** and **age**



Method Overriding



- Now if we want to change this **behavior** and show all **4 attributes** of the Employee i.e. his **name , age ,id** and **salary**, then we will have to **redefine the method __str__()** in our **Emp** class.
- This is called **Method Overriding**
- Thus , **Method Overriding** is a concept in **OOP** which occurs whenever a **derived class redefines the same method** as **inherited** from the **base class**



Modified Example

```
class Person:  
    def __init__(self,age,name):  
        self.age=age  
        self.name=name  
  
    def __str__(self):  
        return f'Age:{self.age},Name:{self.name}'  
  
class Emp(Person):  
    def __init__(self,age,name,id,sal):  
        super().__init__(age,name)  
        self.id=id  
        self.sal=sal  
  
    def __str__(self):  
        return f'Age:{self.age},Name:{self.name},Id:{self.id},Salary:{self.sal}'  
  
e=Emp(24,"Nitin",101,45000)  
print(e)
```

Output:

Age:24,Name:Nitin,Id:101,Salary:45000

Role Of **super()** In Method Overriding



- When we **override** a method of **base class** in the **derived class** then **Python** will always call the **derive's version** of the method.
- But in some cases we also want to call the **base class version** of the **overridden** method.
- In this case we can call the **base class version** of the method from the **derive class** using the function **super()**
- **Syntax:**
super(). <method_name>(<arg>)



Modified Example

```
class Person:  
    def __init__(self,age,name):  
        self.age=age  
        self.name=name  
    def __str__(self):  
        return f'Age:{self.age},Name:{self.name}'  
  
class Emp(Person):  
    def __init__(self,age,name,id,sal):  
        super().__init__(age,name)  
        self.id=id  
        self.sal=sal  
    def __str__(self):  
        str=super().__str__()  
        return f'{str},Id:{self.id},Salary:{self.sal}'  
  
e=Emp(24,"Nitin",101,45000)  
print(e)
```

Output:

Age:24,Name:Nitin,Id:101,Salary:45000

Exercise



- Write a program to create a class called **Circle** having an instance member called **radius**. Provide following methods in **Circle** class
 - **__init__()**: This method should accept an argument and initialize radius with it
 - **area()**: This method should calculate and return Circle's area
- Now create a derived class of **Circle** called **Cylinder** having an instance member called **height**. Provide following methods in **Cylinder** class
 - **__init__()**: This method should initialize instance members **radius** and **height** with the parameter passed.
 - **area()**: This method should override Circle's area() to calculate and return area of Cylinder . (formula: **$2\pi r^2 + 2\pi rh$**)
 - **volume()**: This method should calculate and return Cylinder's volume(formula: **$\pi r^2 h$**)



Solution

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * math.pow(self.radius, 2)
class Cylinder(Circle):
    def __init__(self, radius, height):
        super().__init__(radius)
        self.height = height
    def area(self):
        return 2 * super().area() + 2 * math.pi * self.radius * self.height
    def volume(self):
        return super().area() * self.height
```

Output:

```
Area of cylinder is 1884.9555921538758
Volume of cylinder is 6283.185307179587
```



A Very Important Point!



- Can we call the base class version of an overridden method from outside the derived class ?
- For example , in the previous code we want to call the method **area()** of **Circle** class from our **main script** . How can we do this ?
- Yes this is possible and for this **Python** provides us a special syntax:
- **Syntax:**
`<base_class_name>.<method_name>(<der_obj>)`



Example

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * math.pow(self.radius, 2)
class Cylinder(Circle):
    def __init__(self, radius, height):
        super().__init__(radius)
        self.height = height
    def area(self):
        return 2 * super().area() + 2 * math.pi * self.radius * self.height
    def volume(self):
        return super().area() * self.height
```

```
obj=Cylinder(10,20)
print("Area of cylinder is",obj.area())
print("Volume of cylinder is",obj.volume())
print("Area of Circle:",Circle.area(obj))
```

By calling in this way
we can bypass the
area() method of
Cylinder and directly
call **area()** method of
Circle

Output:

```
Area of cylinder is 1884.9555921538758
Volume of cylinder is 6283.185307179587
Area of Circle: 314.1592653589793
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 44



Today's Agenda

- **Advance Concepts Of Object Oriented Programming-III**
 - MultiLevel Inheritance
 - Hierarchical Inheritance
 - Using The Function `issubclass()`
 - Using The Function `isinstance()`



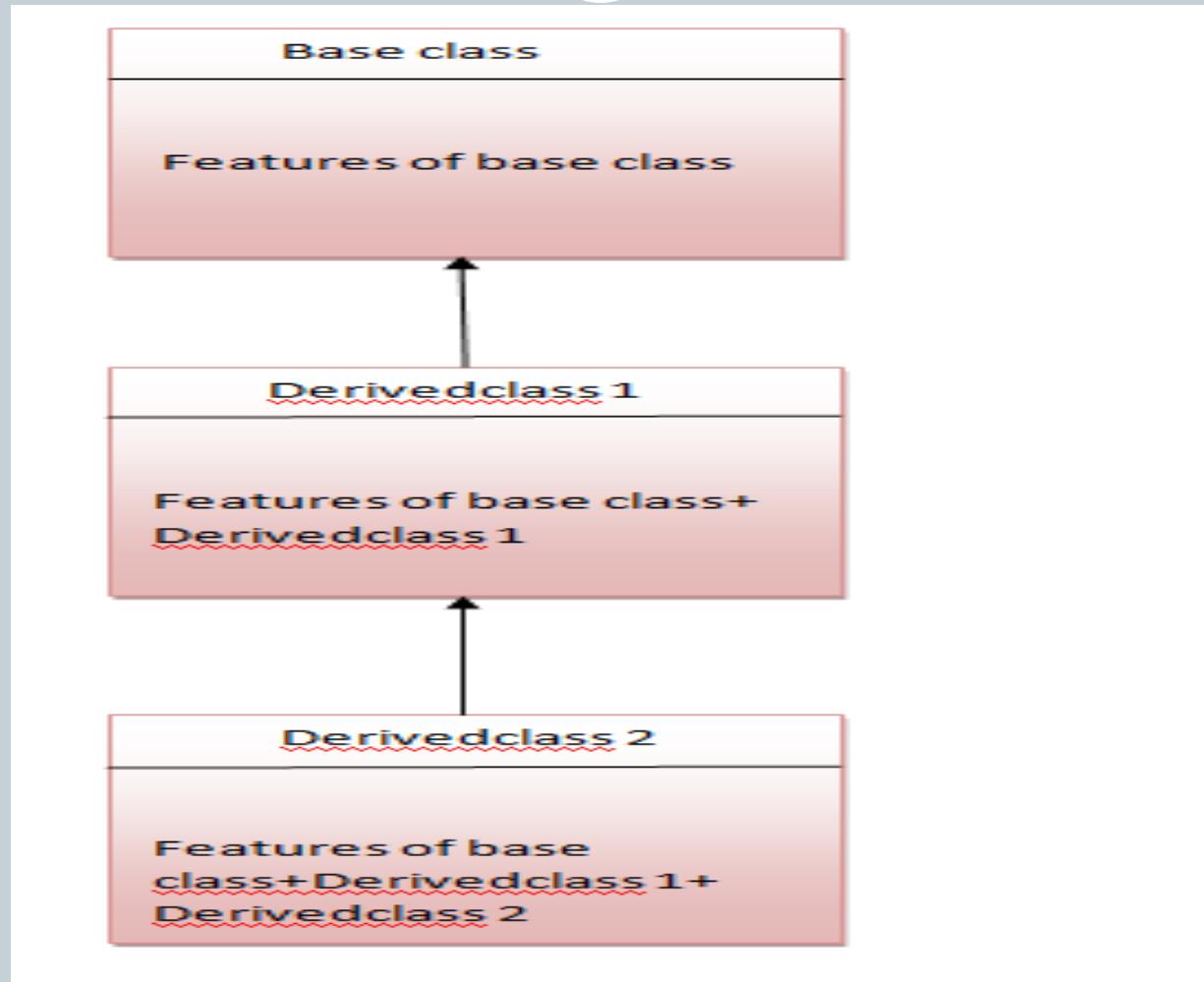
MultiLevel Inheritance



- **Multilevel inheritance** is also possible in Python like other Object Oriented programming languages.
- We can inherit a **derived class** from **another derived class**.
- This process is known as **multilevel inheritance**.
- In Python, **multilevel inheritance** can be done at any depth.



MultiLevel Inheritance





Syntax



class A:

properties of class A

class B(A):

class B inheriting property of class A

more properties of class B

class C(B):

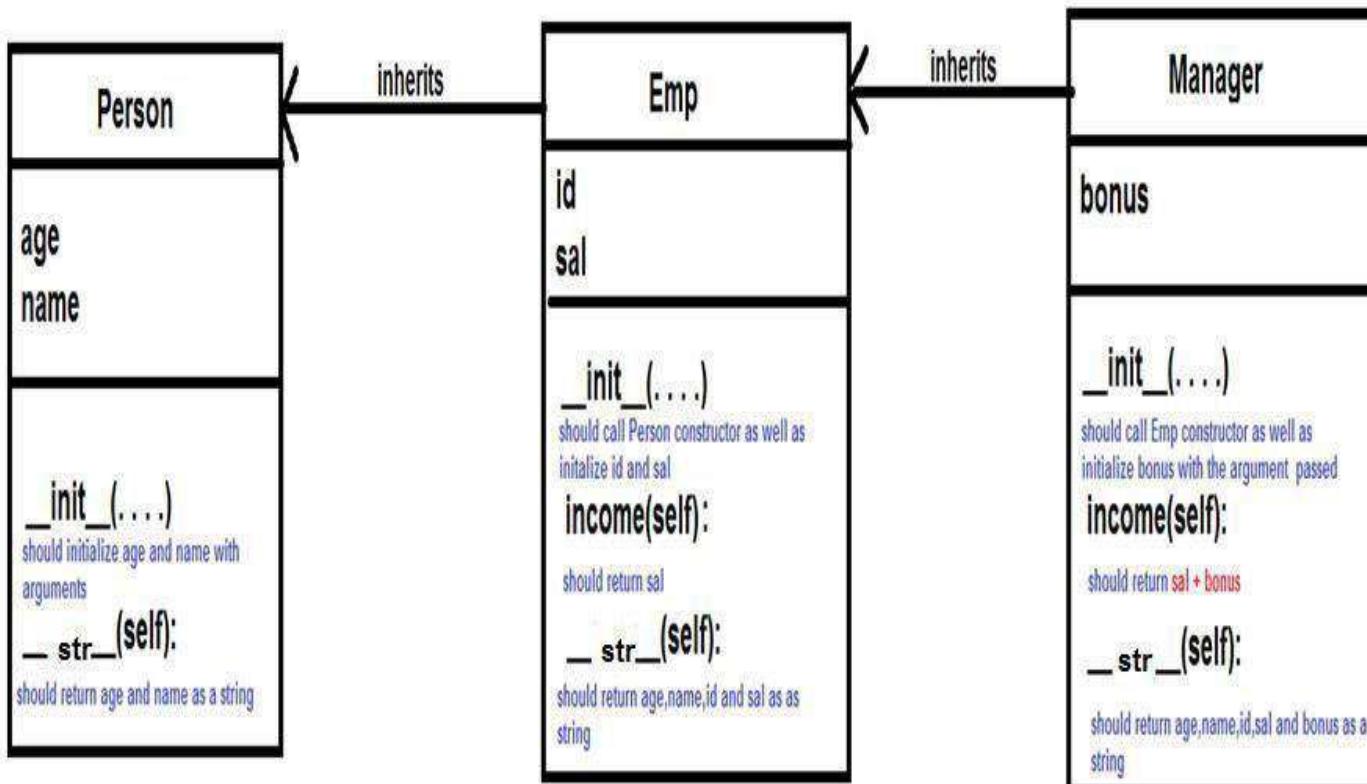
class C inheriting property of class B

thus, class C also inherits properties of class A

more properties of class C

Exercise

- Write a program to create 3 classes **Person** , **Emp** and **Manager**.



Now in the main script create an instance of Manager class and initialize it with required values . Now display 3 things:

1. Complete details of Manager 2. Only the salary of Manager 3. Total income of Manager



Desired Output

```
Person constructor called. . .
Emp constructor called. . .
Manager constructor called. . .
Age:24,Name:Nitin,Id:101,Salary:45000,Bonus:20000
Manager's Salary: 45000
Manager's Total Income: 65000
```



Solution



```
class Person:  
    def __init__(self,age,name):  
        self.age=age  
        self.name=name  
        print("Person constructor called...")  
    def __str__(self):  
        return f'Age:{self.age},Name:{self.name}'  
  
class Emp(Person):  
    def __init__(self,age,name,id,sal):  
        super().__init__(age,name)  
        self.id=id  
        self.sal=sal  
        print("Emp constructor called...")  
    def income(self):  
        return self.sal  
  
    def __str__(self):  
        str=super().__str__()  
        return f'{str},Id:{self.id},Salary:{self.sal}'
```



Solution

```
class Manager(Emp):
    def __init__(self,age,name,id,sal,bonus):
        super().__init__(age,name,id,sal)
        self.bonus=bonus
        print("Manager constructor called...")
    def income(self):
        total=super().income()+self.bonus
        return total
    def __str__(self):
        str=super().__str__()
        return f'{str},Bonus:{self.bonus}'

m=Manager(24,"Nitin",101,45000,20000)
print(m)
print("Manager's Salary:",Emp.income(m))
print("Manager's Total Income:",m.income())
```

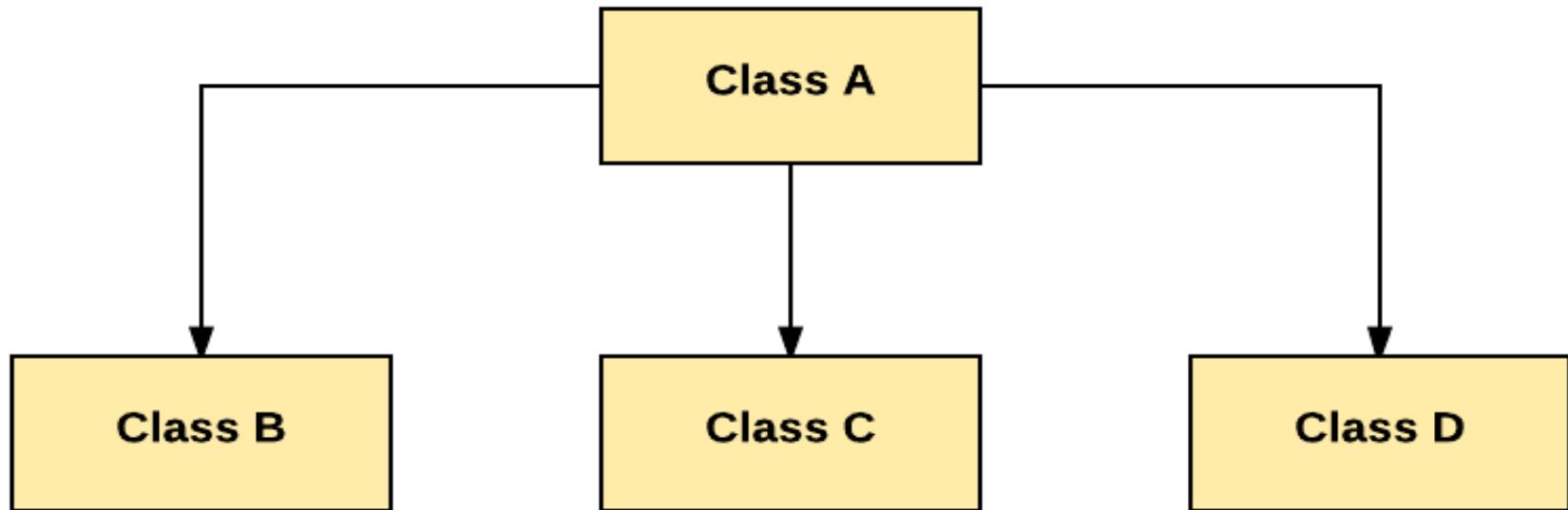
Output:

```
Person constructor called...
Emp constructor called...
Manager constructor called...
Age:24,Name:Nitin,Id:101,Salary:45000,Bonus:20000
Manager's Salary: 45000
Manager's Total Income: 65000
```

Hierarchical Inheritance



- In **Hierarchical Inheritance**, **one class** is inherited by many **sub classes**.





Hierarchical Inheritance



- Suppose you want to write a program which has to keep track of the **teachers** and **students** in a college.
- They have **some common characteristics** such as **name** and **age**.
- They also have specific characteristics such as **salary** for **teachers** and **marks** for **students**.



Hierarchical Inheritance



- One way to solve the problem is that we can create two independent classes for each type and process them.
- But adding a new common characteristic would mean adding to both of these independent classes.
- This quickly becomes very exhaustive task



Hierarchical Inheritance



- A much better way would be to create a common class called **SchoolMember** and then have the **Teacher** and **Student** classes **inherit** from this class
- That is , they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types



Example

```
class SchoolMember:
```

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

```
        self.name = name
```

```
        self.age = age
```

```
        print("Initialized SchoolMember:", self.name)
```

```
    def tell(self):
```

```
        print("Name:", self.name, "Age:", self.age, end=" ")
```



Example

```
class Teacher(SchoolMember):
```

```
    def __init__(self, name, age, salary):  
        super().__init__(name, age)  
        self.salary = salary  
        print("Initialized Teacher:", self.name)
```

```
    def tell(self):  
        super().tell()  
        print("Salary:", self.salary)
```



Example

```
class Student(SchoolMember):
```

```
    def __init__(self, name, age, marks):  
        super().__init__(name, age)  
        self.marks = marks  
        print("Initialized Student:", self.name)
```

```
    def tell(self):
```

```
        super().tell()  
        print("Marks:", self.marks)
```

```
t = Teacher('Mr. Kumar', 40, 80000)
```

```
s = Student('Sudhir', 25, 75)
```

```
print()
```

```
members = [t, s]
```

```
for member in members:
```

```
    member.tell()
```

Output

```
Initialized SchoolMember: Mr. Kumar  
Initialized Teacher: Mr. Kumar  
Initialized SchoolMember: Sudhir  
Initialized Student: Sudhir
```

```
Name: Mr. Kumar Age: 40 Salary: 80000  
Name: Sudhir Age: 25 Marks: 75
```

How To Check Whether A Class Is A SubClass Of Another ?



- **Python** provides a function **issubclass()** that directly tells us if a class is a **subclass** of **another class**.
- **Syntax:**

issubclass(<name of der class>, <name of base class>)

- The **function** returns **True** if the **classname** passed as **first argument** is the derive class of the **classname** passed as **second argument** otherwise it returns **False**



Guess The Output ?



```
class MyBase(object):  
    pass
```

```
class MyDerived(MyBase):  
    pass
```

```
print(issubclass(MyDerived, MyBase))  
print(issubclass(MyBase, object))  
print(issubclass(MyDerived, object))  
print(issubclass(MyBase, MyDerived))
```

Output:

```
True  
True  
True  
False
```



Guess The Output ?

```
class MyBase:  
    pass
```

```
class MyDerived(MyBase):  
    pass
```

```
print(issubclass(MyDerived, MyBase))  
print(issubclass(MyBase, object))  
print(issubclass(MyDerived, object))  
print(issubclass(MyBase, MyDerived))
```

Output:

```
True  
True  
True  
False
```

In **Python 3** , every class implicitly **inherits** from **object** class but in **Python 2** it is not so. Thus in **Python 2** the **2nd** and **3rd** **print()** statements would return **False**



Alternate Way



- Another way to do the same task is to call the function **isinstance()**
- **Syntax:**

isinstance(<name of obj ref>, <name of class>)

- The **function** returns **True** if the **object reference** passed as **first argument** is an instance of the **classname** passed as **second argument** or any of its **subclasses**. Otherwise it returns **False**



Guess The Output ?



```
class MyBase:
```

```
    pass
```

```
class MyDerived(MyBase):
```

```
    pass
```

```
d = MyDerived()
```

```
b = MyBase() print(isinstance(d,
```

```
MyBase)) print(isinstance(d,
```

```
MyDerived))
```

```
print(isinstance(d, object))
```

```
print(isinstance(b, MyBase))
```

```
print(isinstance(b, MyDerived))
```

```
print(isinstance(b, object))
```

Output:

```
True
```

```
True
```

```
True
```

```
True
```

```
False
```

```
True
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 45



Today's Agenda

- **Advance Concepts Of Object Oriented Programming-IV**
 - Multiple Inheritance
 - The MRO Algorithm
 - Hybrid Inheritance
 - The Diamond Problem



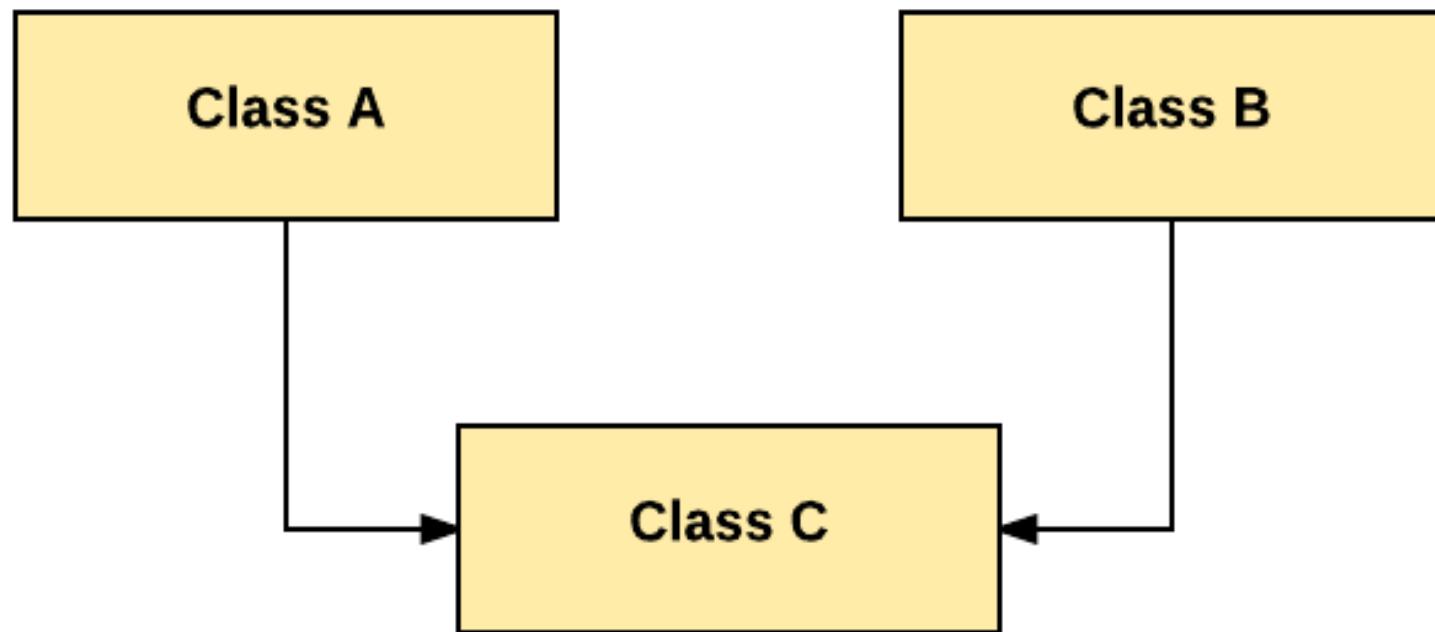
Multiple Inheritance



- Like **C++**, in **Python** also a class can be derived from more than one base class.
- This is called **multiple inheritance**.
- In **multiple inheritance**, the features of all the base classes are inherited into the derived class.



Multiple Inheritance





Syntax



class A:

properties of class A

class B:

#properties of class B

class C(A,B):

class C inheriting property of class A

class C inheriting property of class B

more properties of class C



Example

```
class Person:  
    def __init__(self,name,age):  
        self.name=name  
        self.age=age  
  
    def getname(self):  
        return self.name  
  
    def getage(self):  
        return self.age  
  
class Student:  
    def __init__(self,roll,per):  
        self.roll=roll  
        self.per=per  
  
    def getroll(self):  
        return self.roll  
  
    def getper(self):  
        return self.per
```

```
class ScienceStudent(Person,Student):  
  
    def __init__(self,name,age,roll,per,stream):  
        Person.__init__(self,name,age)  
        Student.__init__(self,roll,per)  
        self.stream=stream  
  
    def getstream(self):  
        return self.stream  
  
ms=ScienceStudent("Suresh",19,203,89.4,"maths")  
print("Name:",ms.getname())  
print("Age:",ms.getage())  
print("Roll:",ms.getroll())  
print("Per:",ms.getper())  
print("Stream:",ms.getstream())
```

Output:

```
Name: Suresh  
Age: 19  
Roll: 203  
Per: 89.4  
Stream: maths
```



Guess The Output ?



```
class A:  
    def m(self):  
        print("m of A called")  
  
class B:  
    def m(self):  
        print("m of B called")  
  
class C(A,B):  
    pass
```

Output:

m of A called

Why did m() of A
got called ?

This is because of a
special rule in
Python called MRO

obj=C()
obj.m()



What Is MRO In Python ?



- In languages that use **multiple inheritance**, the order in which **base classes** are searched when looking for a **method** is often called the **Method Resolution Order**, or **MRO**.
- **MRO RULE :**
 - In the multiple inheritance scenario, any specified attribute is searched **first in the current class**. If not found, the search continues into **parent classes, left-right fashion** and **then in depth-first without searching same class twice**.



Can We See This MRO ?



- Yes, Python allows us to see this MRO by calling a method called **mro()** which is present in every class by default.



Example



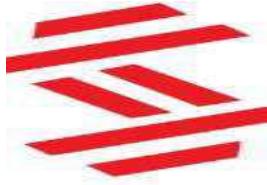
```
class A:  
    def m(self):  
        print("m of A called")
```

```
class B:  
    def m(self):  
        print("m of B called")
```

```
class C(A,B):  
    pass  
print(C.mro())
```

Output

```
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```



Another Way To See MRO ?



- There is a tuple also called `__mro__` made available in **every class** by **Python** using which we can get the same output as before



Example



```
class A:  
    def m(self):  
        print("m of A called")
```

```
class B:  
    def m(self):  
        print("m of B called")
```

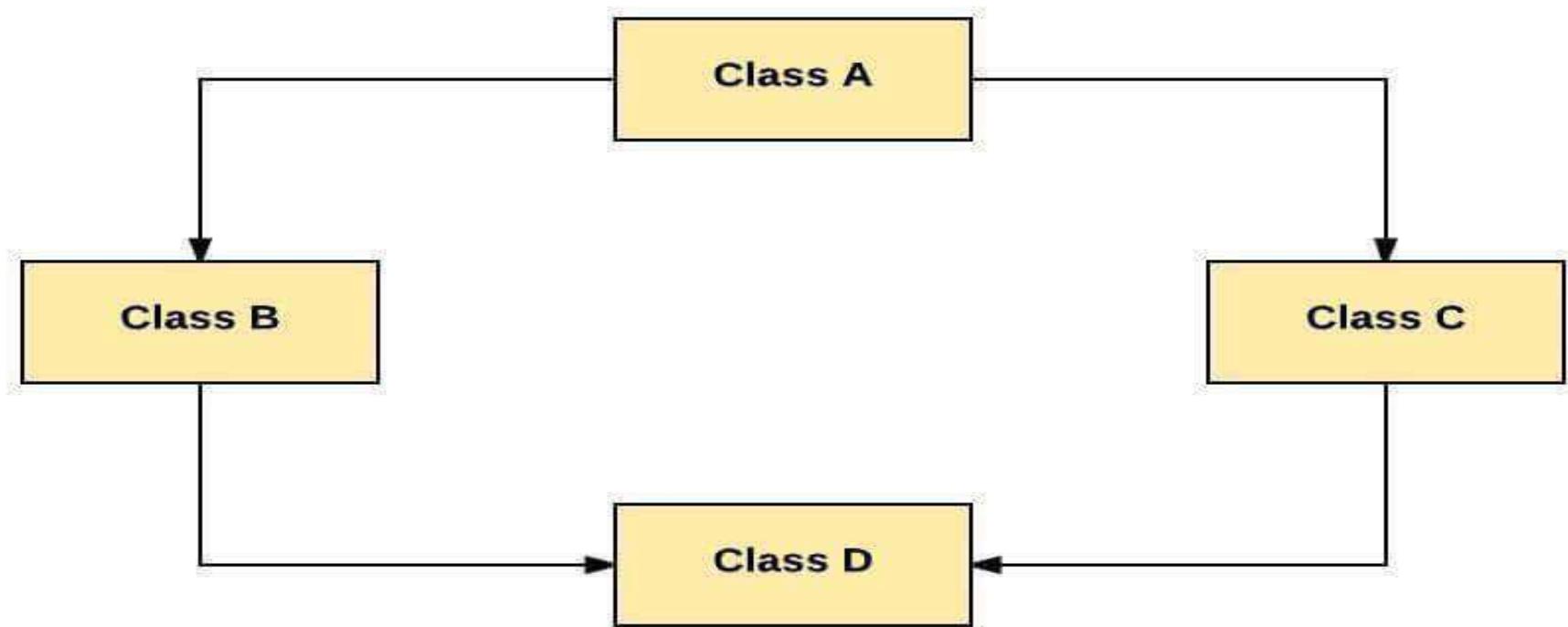
```
class C(A,B):  
    pass  
print(C.__mro__)
```

Output

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

The Hybrid Inheritance

- This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.



Example

```
class A:  
    def m1(self):  
        print("m1 of A called")  
  
class B(A):  
    def m2(self):  
        print("m2 of B called")  
  
class C(A):  
    def m3(self):  
        print("m3 of C called")  
  
class D(B,C):  
    pass
```

obj=D()
obj.m1()
obj.m2()
obj.m3()

Output:

```
m1 of A called  
m2 of B called  
m3 of C called
```



The Diamond Problem



- The “**diamond problem**” is the generally used term for an **ambiguity** that arises in **hybrid inheritance** .
- Suppose two classes **B** and **C** inherit from a superclass **A**, and another class **D** inherits from both **B** and **C**.
- If there is a **method** "m" in **A** that **B** and **C** have overridden, then the question is **which version of the method does D inherit?**



Guess The Output

```
class A:  
    def m(self):  
        print("m of A called")
```

```
class B(A):  
    def m(self):  
        print("m of B called")
```

```
class C(A):  
    def m(self):  
        print("m of C called")
```

```
class D(B,C):  
    pass
```

Output:

m of B called

obj=D()
obj.m()

Why m() of B was called ?

As discussed previously , Python uses MRO to search for an attribute which goes from left to right and then in depth first.

Now since B is the first inherited class of D so Python called m() of B



Guess The Output



```
class A:  
    def m(self):  
        print("m of A called")
```

obj=D()
obj.m()

```
class B(A):  
    def m(self):  
        print("m of B called")
```

```
class C(A):  
    def m(self):  
        print("m of C called")
```

```
class D(C,B):  
    pass
```

Output:

m of C called



Guess The Output

```
class A:  
    def m(self):  
        print("m of A called")
```

```
class B(A):  
    pass
```

```
class C(A):  
    def m(self):  
        print("m of C called")
```

```
class D(B,C):  
    pass
```

Output:
m of C called

obj=D()
obj.m()

Why m() of C was called ?

MRO goes from left to right first and then depth first. In our case Python will look for method m() in B but it won't find it there.

Then it will search m() in C before going to A. Since it finds m() in C, it executes it dropping the further search



Guess The Output



class A:

```
def m(self):  
    print("m of A called")
```

obj=D()
obj.m()

class B(A):

```
def m(self):  
    print("m of B called")
```

class C(A):

```
def m(self):  
    print("m of C called")
```

class D(B,C):

```
def m(self):  
    print("m of D called")
```

Output:

m of D called



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 46



Today's Agenda



• **Exception Handling**

- Introduction To Exception Handling
- Exception Handling Keywords
- Exception Handling Syntax
- Handling Multiple Exceptions
- Handling All Exceptions



What Is An Exception ?



- **Exception** are errors that occur at runtime .
- In other words , if our program encounters an **abnormal situation** during it's execution it **raises** an **exception**.
- **For example**, the statement
a=10/0
will generate an **exception** because **Python** has no way to solve **division by 0**

What Python Does When An Exception Occurs ?



- Whenever an **exception** occurs , **Python** does 2 things :
 - It immediately **terminates** the code
 - It displays the **error message** related to the exception in a **technical way**
- Both the steps taken by **Python** cannot be considered user friendly because
 - Even if a statement generates exception , still other parts of the program must get a chance to run
 - The error message must be simpler for the user to understand



A Sample Code

```
a=int(input("Enter first no:"))
b=int(input("Enter second no:"))
c=a/b
print("Div is",c)
d=a+b
print("Sum is",d)
```

Output:

```
Enter first no:10
Enter second no:5
Div is 2.0
Sum is 15
```

```
Enter first no:10
Enter second no:0
Traceback (most recent call last):
  File "except1.py", line 3, in <module>
    c=a/b
ZeroDivisionError: division by zero
```

As we can observe , in the second run the code generated exception because Python does not know how to handle division by 0. Moreover it did not even calculated the sum of 10 and 0 which is possible



A Sample Code

```
a=int(input("Enter first no:"))
b=int(input("Enter second no:"))
c=a/b
print("Div is",c)
d=a+b
print("Sum is",d)
```

Output:

```
Enter first no:10
Enter second no:2a
Traceback (most recent call last):
  File "except1.py", line 2, in <module>
    b=int(input("Enter second no:"))
ValueError: invalid literal for int() with base 10: '2a'
```

In this case since it is not possible for Python to convert “2a” into an integer , so it generated an exception . But the message it displays is too technical to understand

How To Handle Such Situations ?



- If we want our program to behave **normally**, even if an **exception** occurs, then we will have to apply **Exception Handling**
- **Exception handling** is a mechanism which allows us to handle errors **gracefully** while the program is running instead of **abruptly ending** the program execution.



Exception Handling Keywords



- Python provides **5 keywords** to perform **Exception Handling**:
 - **try**
 - **except**
 - **else**
 - **raise**
 - **finally**



Exception Handling Syntax



- Following is the **syntax** of a **Python try-except-else** block.

try:

You do your operations here;

.....

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

.....

else:

If there is no exception then execute this block.

Remember !

In place of **Exception I** and **Exception II**, we have to use the names of **Exception classes** in Python

Improved Version Of Previous Code



```
a=int(input("Enter first no:"))
b=int(input("Enter second no:"))

try:
    c=a/b
    print("Div is",c)
except ZeroDivisionError:
    print("Denominator should not be 0")
d=a+b

print("Sum is",d)
```

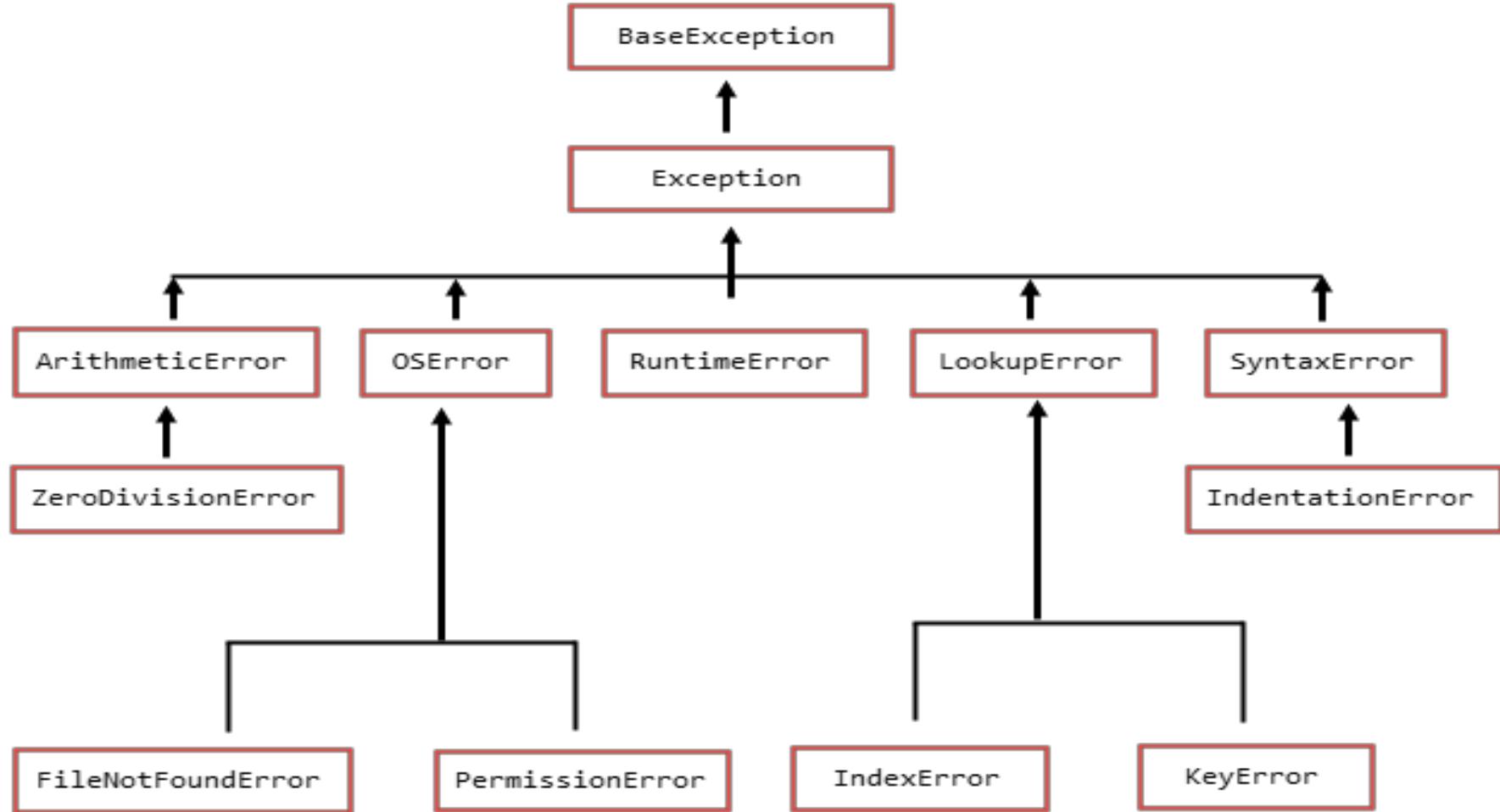
Output:

```
Enter first no:10
Enter second no:0
Denominator should not be 0
Sum is 10
```

```
Enter first no:10
Enter second no:3
Div is 3.333333333333335
Sum is 13
```



Exception Hierarchy





Important Exception Classes

Exception Class	Description
Exception	Base class for all exceptions
ArithmeticError	Raised when numeric calculations fails
FloatingPointError	Raised when a floating point calculation fails
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types
OverflowError	Raised when result of an arithmetic operation is too large to be represented
ImportError	Raised when the imported module is not found in Python version < 3.6
ModuleNotFoundError	Raised when the imported module is not found from Python version >=3.6



Important Exception Classes

Exception Class	Description
LookupError	Raised when searching /lookup fails
KeyError	Raised when the specified key is not found in the dictionary
IndexError	Raised when index of a sequence is out of range
NameError	Raised when an identifier is not found in the local or global namespace
UnboundLocalError	Raise when we use a local variable in a function before declaring it.
TypeError	Raised when a function or operation is applied to an object of incorrect type
ValueError	Raised when a function gets argument of correct type but improper value



Important Exception Classes

Exception Class	Description
AttributeError	Raised when a non-existent attribute is referenced.
OSError	Raised when system operation causes system related error.
FileNotFoundException	Raised when a file is not present
FileExistsError	Raised when we try to create a directory which is already present
PermissionError	Raised when trying to run an operation without the adequate access rights.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.



Handling Multiple Exception



- A **try** statement may have more than one **except** clause for **different exceptions**.
- But at most one **except** clause will be executed



Point To Remember



- Also , we must remember that if we are handling **parent and child exception classes** in **except** clause then the **parent exception** must appear **after child exception** , otherwise child except will never get a chance to run



Guess The Output !

```
import math
```

```
try:
```

```
    x=10/5
```

```
    print(x)
```

```
    ans=math.exp(3)
```

```
    print(ans)
```

```
except ZeroDivisionError:
```

```
    print("Division by 0 exception occurred!")
```

```
except ArithmeticError:
```

```
    print("Numeric calculation failed!")
```

Output:

2.0

20.085536923187668



Guess The Output !

```
import math
```

```
try:
```

```
    x=10/0
```

```
    print(x)
```

```
    ans=math.exp(20000)
```

```
    print(ans)
```

```
except ZeroDivisionError:
```

```
    print("Division by 0 exception occurred!")
```

```
except ArithmeticError:
```

```
    print("Numeric calculation failed!")
```

Output:

Division by 0 exception occurred!



Guess The Output !

```
import math
```

```
try:
```

```
    x=10/5
```

```
    print(x)
```

```
    ans=math.exp(20000)
```

```
    print(ans)
```

```
except ZeroDivisionError:
```

```
    print("Division by 0 exception occurred!")
```

```
except ArithmeticError:
```

```
    print("Numeric calculation failed!")
```

Output:

2.0

Numeric calculation failed!



Guess The Output !

```
import math
```

```
try:
```

```
    x=10/5
```

```
    print(x)
```

```
    ans=math.exp(20000)
```

```
    print(ans)
```

```
except ArithmeticError:
```

```
    print("Numeric calculation failed!")
```

```
except ZeroDivisionError:
```

```
    print("Division by 0 exception occurred!")
```

Output:

```
2.0
```

```
Numeric calculation failed!
```



Guess The Output !

```
import math
```

```
try:
```

```
    x=10/0
```

```
    print(x)
```

```
    ans=math.exp(20000)
```

```
    print(ans)
```

```
except ArithmeticError:
```

```
    print("Numeric calculation failed!")
```

```
except ZeroDivisionError:
```

```
    print("Division by 0 exception occurred!")
```

Output:

```
Numeric calculation failed!
```



Exercise



- Write a program to ask the user to input 2 integers and calculate and print their division. Make sure your program behaves as follows:
 - If the user enters a non integer value then ask him to enter only integers
 - If denominator is 0 , then ask him to input non-zero denominator
 - Repeat the process until correct input is given
- Only if the inputs are correct then display their division and terminate the code



Sample Output

```
Input first no:10
Input second no:0
Please input non-zero denominator
Input first no:a
Please input integers only! Try again
Input first no:10
Input second no:a
Please input integers only! Try again
Input first no:4
Input second no:5
Div is 0.8
```



Solution



```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break
    except ValueError:
        print("Please input integers only! Try again")
    except ZeroDivisionError:
        print("Please input non-zero denominator")
```

Single **except**, Multiple Exception



- If we want to write a single **except** clause to handle **multiple exceptions** , we can do this .
- For this we have to write **names of all the exceptions** within **parenthesis** separated with **comma** after the keyword **except**



Example

while(True):

try:

a=int(input("Input first no:"))

b=int(input("Input second no:"))

c=a/b

print("Div is ",c)

break

except (ValueError,ZeroDivisionError):

print("Either input is incorrect or denominator is 0. Try again!")



Sample Output



```
Input first no:4
Input second no:0
Either input is incorrect or denominator is 0. Try again!
Input first no:10
Input second no:bhopal
Either input is incorrect or denominator is 0. Try again!
Input first no:10
Input second no:4
Div is 2.5
```



Handling All Exceptions



- We can write the keyword **except** without any **exception class name** also .
- In this case for every **exception** this except clause will run .
- The only problem will be that we will never know the **type of exception** that has occurred!



Exception Handling Syntax

- Following is the **syntax** of a **Python handle all exception** block.

try:

You do your operations here

.....

except :

For every kind of exception this block will execute

Notice , we have not provided any name for the exception



Example

```
while(True):  
    try:  
        a=int(input("Input first no:"))  
        b=int(input("Input second no:")) c=a/b  
        print("Div is ",c) break  
    except:  
        print("Some problem occurred. Try again!")
```



Sample Output



Input first no:10

Input second no:0

Some problem occurred. Try again!

Input first no:10

Input second no:a

Some problem occurred. Try again!

Input first no:10

Input second no:4

Div is 2.5

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 47





Today's Agenda



• **Exception Handling**

- Using Exception Object
- Getting Details Of Exception
- Raising An Exception
- Using finally Block
- Creating User Defined Exceptions



Using Exception Object

- Now we know how to handle exception, in this section we will learn how to access **exception object** in exception handler code.
- To access the **exception object** created by Python we can use the keyword **as** and assign it to a **variable**.
- Finally using that variable we can get the details of the exception



Example

```
while(True):
```

```
    try:
```

```
        a=int(input("Input first no:"))
```

```
        b=int(input("Input second no:"))
```

```
        c=a/b
```

```
        print("Div is ",c)
```

```
    break;
```

```
except (ValueError,ZeroDivisionError) as e:
```

```
    print(e)
```



Sample Output



```
Input first no:10
```

```
Input second no:0
```

```
division by zero
```

```
Input first no:10
```

```
Input second no:a
```

```
invalid literal for int() with base 10: 'a'
```

```
Input first no:10
```

```
Input second no:5
```

```
Div is 2.0
```

Obtaining Exception Details Using traceback class



- Sometimes , we need to print the details of the exception exactly ***like Python does*** .
- We do this normally , when we are **debugging our code**.
- The module **traceback** helps us do this

Obtaining Exception Details Using traceback module



- This module contains a function called **format_exc()**
- It returns **complete details** of the exception as a **string**.
- This **string** contains:
 - The **program name** in which **exception** occurred
 - **Line number** where **exception** occurred
 - The **code** which generated the **exception**
 - The **name** of the **exception class**
 - The **message** related to the **exception**



Example

```
import traceback
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break
    except:
        print(traceback.format_exc())
```



Sample Output



```
Input first no:10
```

```
Input second no:0
```

```
Traceback (most recent call last):
```

```
  File "except5.py", line 6, in <module>
    c=a/b
```

```
ZeroDivisionError: division by zero
```

```
Input first no:10
```

```
Input second no:bhopal
```

```
Traceback (most recent call last):
```

```
  File "except5.py", line 5, in <module>
    b=int(input("Input second no:"))
```

```
ValueError: invalid literal for int() with base 10: 'bhopal'
```

```
Input first no:10
```

```
Input second no:5
```

```
Div is 2.0
```



Raising An Exception



- We can force **Python** to generate an **Exception** using the keyword **raise**.
- This is normally done in those situations where we want **Python** to throw an exception in a particular condition of our choice
- **Syntax:**
 - `raise ExceptionClassName`
 - `raise ExceptionClassName(message)`



Exercise



- Write a program to ask the user to input 2 integers and calculate and print their division. Make sure your program behaves as follows:
 - If the user enters a non integer value then ask him to enter only integers
 - If denominator is 0 , then ask him to input non-zero denominator
 - **If any of the numbers is negative or numerator is 0 then display the message negative numbers not allowed**
 - Repeat the process until correct input is given
- Only if the inputs are correct then display their division and terminate the code



Sample Output

Input first no:10

Input second no:-4

Negative numbers not allowed! Try again

Input first no:10

Input second no:0

Please input non-zero denominator

Input first no:-1

Input second no:4

Negative numbers not allowed! Try again

Input first no:10

Input second no:bhopal

Please input integers only! Try again

Input first no:20

Input second no:5

Div is 4.0



Solution



```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        if a<=0 or b<0:
            raise Exception("Negative numbers not allowed!Try again")
        c=a/b
        print("Div is ",c)
        break;
    except ValueError:
        print("Please input integers only! Try again")
    except ZeroDivisionError:
        print("Please input non-zero denominator")
    except Exception as e:
        print(e)
```



The **finally** Block



- If we have a code which we want to run in all situations, then we should write it inside the **finally** block.
- **Python** will always run the instructions coded in the **finally** block.
- It is the most common way of doing **clean up tasks** , like, **closing a file** or **disconnecting with the DB** or **logging out the user** etc



Syntax Of The **finally** Block



- The **finally** block has 2 syntaxes:

Syntax 1

```
try:  
    # some exception generating code  
except:  
    # exception handling code  
finally:  
    # code to be always executed
```

Syntax 2

```
try:  
    # some exception generating code  
finally:  
    # code to be always executed
```



Guess The Output ?

```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break;
    except ZeroDivisionError:
        print("Denominator should not be zero")
    finally:
        print("Thank you for using the app!")
```

Output:

```
Input first no:10
Input second no:0
Denominator should not be zero
Thank you for using the app!
Input first no:10
Input second no:5
Div is 2.0
Thank you for using the app!
```

```
Input first no:10
Input second no:a
Traceback (most recent call last):
  File "except8.py", line 4, in <module>
    b=int(input("Input second no:"))
ValueError: invalid literal for int() with base 10: 'a'
```

Creating User Defined Exception



- Python has many **built-in exceptions** which forces our program to output an error when something in it goes wrong.
- However, sometimes we may need to create our own exceptions which will be more suitable for our purpose.
- Such exceptions are called **User Defined Exceptions**

Creating User Defined Exception



- In **Python**, users can define such exceptions by creating a **new class**.
- This **exception class** has to be **derived**, either directly or indirectly, from **Exception** class.
- Most of the **built-in exceptions** are also derived form this class.

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



PYTHON

LECTURE 48



Today's Agenda



• **Operator Overloading**

- What Is Operator Overloading
- How To Perform Operator Overloading
- List Of Operators Which Can Be Overloaded
- Reverse Arithmetic Operators

What Is Operator Overloading?



- **Operator overloading** means **redefining** existing operators in **Python** to work on objects of **our classes**.
- For example, a **+** operator is used to **add** the **numeric values** as well as to **concatenate** the **strings**.
- That's because operator **+** is overloaded for **int** class and **str** class.

What Is Operator Overloading?



- But we can give **extra functionality** to this **+** operator and use it with **objects** of **our own class**.
- This method of giving **extra functionality** to the operators is called **operator overloading**.



Guess The Output ?

```
class Point:  
    def __init__(self,x,y):  
        self.x=x  
        self.y=y  
    def __str__(self):  
        return f'x={self.x},y={self.y}'  
  
p1=Point(10,20)  
p2=Point(30,40)  
p3=p1+p2  
print(p3)
```

Output:

```
Traceback (most recent call last):  
  File "opov11.py", line 10, in <module>  
    p3=p1+p2  
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Why did **TypeError** occur?

TypeError was raised since Python didn't know how to add two **Point** objects together.

How To Perform Operator Overloading?



- There is an **underlying mechanism** related to **operators** in Python.
- The thing is when we use **operators**, a **special function** or **magic function** is automatically invoked that is associated with that **particular operator**.

How To Perform Operator Overloading?



- For example, when we use **+ operator**, the magic method **__add__** is automatically invoked in which the operation for **+ operator** is defined.
- So **by changing this magic method's code**, we can give extra meaning to the **+ operator**.



Example

```
class Point:  
    def __init__(self,x,y):  
        self.x=x  
        self.y=y  
    def __add__(self,other):  
        x=self.x+other.x  
        y=self.y+other.y  
        p=Point(x,y)  
        return p  
  
    def __str__(self):  
        return f'x={self.x},y={self.y}'  
p1=Point(10,20)  
p2=Point(30,40)  
p3=p1+p2  
print(p3)  
Output:  
x=40 , y=60
```



Explanation



- When we wrote **p1 + p2**, then **Python** did the following:
 - It searched for the magic method `__add__()` in our **Point** class since the left side operand i.e. **p1** is of **Point** class.
 - After finding `__add__()` in our class **Python converted** our statement **p1+p2** to **p1.__add__(p2)** which in turn is **Point.__add__(p1,p2)**.
 - So **p1** is passed as **self** and **p2** is passed to **other**
 - Finally **addition was done** and a new object **p** was returned which was copied to **p3**

Guess The Output



```
class Point:  
    def __init__(self,x,y):  
        self.x=x  
        self.y=y  
    def __add__(self,other):  
        x=self.x+other.x  
        y=self.y+other.y  
        p=Point(x,y)  
        return p  
  
    def __str__(self):  
        return f'x={self.x},y={self.y}'
```

```
p1=Point(10,20)  
p2=Point(30,40)  
p3=p1+p2  
print(p3)  
p4=p1+p2+p3  
print(p4)
```

Output:

```
x=40 , y=60  
x=80 , y=120
```

These Notes Have Python_world_wide Rights
Acts of Government of India. So commercial Use of this notes is Strictly prohibited





Exercise



- Write a program to create a class called **Distance** having **2 instance members** called **feet** and **inches** . Provide following methods in **Distance** class
 - **__init__()** : This method should accept 2 arguments and initialize **feet** and **inches** with it
 - **__str__()**: This method should return string representation of **feet** and **inches**
 - **__add__()** : This method should add 2 **Distance objects** and return another **Distance object** as the result. While adding if sum of **inches** becomes **>=12** then it should be appropriately converted to **feet**



Solution

```
class Distance:  
    def __init__(self,feet,inches):  
        self.feet=feet  
        self.inches=inches  
    def __add__(self,other):  
        feet=self.feet+other.feet  
        inches=self.inches+other.inches  
        if inches>=12:  
            feet=feet+inches//12  
            inches=inches%12  
        d=Distance(feet,inches)  
        return d  
  
    def __str__(self):  
        return f'feet={self.feet},inches={self.inches}'
```

Output:

```
feet=10,inches=6  
feet=8,inches=9  
feet=19,inches=3
```



Guess The Output ?

```
class Distance:  
    def __init__(self,feet,inches):  
        self.feet=feet  
        self.inches=inches  
    def __add__(self,other):  
        feet=self.feet+other.feet  
        inches=self.inches+other.inches  
        if inches>=12:  
            feet=feet+inches//12  
            inches=inches%12  
        d=Distance(feet,inches)  
        return d  
  
    def __str__(self):  
        return f"feet={self.feet},inches={self.inches}"
```

Output:

```
feet=10,inches=6  
feet=8,inches=9  
feet=19,inches=3  
Traceback (most recent call last):  
  File "opov12.py", line 23, in <module>  
    d4=d1+10  
  File "opov12.py", line 6, in __add__  
    feet=self.feet+other.feet  
AttributeError: 'int' object has no attribute 'feet'
```

```
d1=Distance(10,6)  
d2=Distance(8,9)  
d3=d1+d2  
print(d1)  
print(d2)  
print(d3)  
d4=d1+10  
print(d4)
```

Why did **AttributeError** occur ?

This is because **Python** is trying to use the **int** object as **Distance** object and since **int** class has no **feet** data member the code is throwing **AttributeError**



Solution

```
class Distance:  
    def __init__(self,feet,inches):  
        self.feet=feet  
        self.inches=inches  
    def __add__(self,other):  
        if isinstance(other,Distance):  
            feet=self.feet+other.feet  
            inches=self.inches+other.inches  
        else:  
            feet=self.feet+other  
            inches=self.inches+other  
        if inches>=12:  
            feet=feet+inches//12  
            inches=inches%12  
        d=Distance(feet,inches)  
        return d  
    def __str__(self):  
        return f"feet={self.feet},inches={self.inches}"
```

Output:

```
feet=10 , inches=6  
feet=8 , inches=9  
feet=19 , inches=3  
feet=21 , inches=4
```

```
d1=Distance(10,6)  
d2=Distance(8,9)  
d3=d1+d2  
print(d1)  
print(d2)  
print(d3)  
d4=d1+10  
print(d4)
```

We have used **isinstance()** function to determine whether the argument **other** is of type **Distance** or not . If it is of type **Distance** we perform usual addition logic , otherwise we simply add the argument **other** to **self.feet** and **self.inches** as **int** value

List Of Arithmetic Operator For Overloading



Addition	$p1 + p2$	<code>p1.__add__(p2)</code>
Subtraction	$p1 - p2$	<code>p1.__sub__(p2)</code>
Multiplication	$p1 * p2$	<code>p1.__mul__(p2)</code>
Power	$p1 ** p2$	<code>p1.__pow__(p2)</code>
Division	$p1 / p2$	<code>p1.__truediv__(p2)</code>
Floor Division	$p1 // p2$	<code>p1.__floordiv__(p2)</code>
Remainder (modulo)	$p1 \% p2$	<code>p1.__mod__(p2)</code>



Exercise



- Write a program to create a class called **Book** having **2 instance members** called **name** and **price** . Provide following methods in **Book** class
 - **__init__()** : This method should accept 2 arguments and initialize **name** and **price** with it
 - **__str__()**: This method should return string representation of **name** and **price**
 - **__add__()** : This method should add price of 2 **Books** and return the **total price**



Solution

```
class Book:  
    def __init__(self,name,price):  
        self.name=name  
        self.price=price  
  
    def __add__(self,other):  
        totalprice=self.price+other.price  
        return totalprice  
  
    def __str__(self):  
        return f'Name={self.name}, Price={self.price}'
```

```
b1=Book("Mastering Python",300)  
b2=Book("Mastering Java",500)  
print(b1)  
print(b2)  
print("Total price of books is:",b1+b2)
```

Output:

```
name=Mastering Python, price=300  
name=Mastering Java, price=500  
Total price of books is: 800
```



Guess The Output ?

```
class Book:  
    def __init__(self,name,price):  
        self.name=name  
        self.price=price  
  
    def __add__(self,other):  
        totalprice=self.price+other.price  
        return totalprice  
  
    def __repr__(self):  
        return f'Name={self.name}, Price={self.price}'
```

```
b1=Book("Mastering Python",300)  
b2=Book("Mastering Java",500)  
b3=Book("Mastering C++",400)  
print(b1)  
print(b2)  
print(b3)  
print("Total price of books  
is:",b1+b2+b3)
```

Output:

```
name=Mastering Python, price=300  
name=Mastering Java, price=500  
name=Mastering C++, price=400  
Traceback (most recent call last):  
  File "opov14.py", line 19, in <module>  
    print("Total price of books is:",b1+b2+b3)  
TypeError: unsupported operand type(s) for +: 'int' and 'Book'
```



Why Did TypeError Occur ?



- **TypeError** occurred because **Python** evaluated the statement **b1+b2+b3** as follows:
 - At first it solved **b1+b2** , which became **b1.__add__(b2)**.
 - So Python called **__add__()** method of **Book** class since the **left operand** is **b1** which is object of class **Book**
 - This call returned the **total price** of **b1** and **b2** which is **800**.
 - Now **Python** used **800** as the **calling object** and **b3** as argument so the call became **800.__add__(b3)**.
 - So Python now looks for a method **__add__()** in **int** class which can add an **int** and a **book** but it could not find such a method in **int** class which can take **Book** object as argument .
 - So the code threw **TypeError**

What Is The Solution To This Problem ?



- The solution to this problem is to provide **reverse special methods** in our class.
- The standard methods like `__add__()`, `__sub__()` only work when we have **object** of our class as **left operand** .

What Is The Solution To This Problem ?



- But they don't work when we have **object** of our class on **right side of the operator** and **left side operand** is not the **instance** of our class.
- **For example :** `obj+10` will call `__add__()` internally, but `10+obj` will not call `__add__()`

What Is The Solution To This Problem ?



- Therefore, to help us make our classes mathematically correct, Python provides us with **reverse/reflected special methods** such as `__radd__()`, `__rsub__()`, `__rmul__()`, and so on.
- These handle calls such as `x + obj`, `x - obj`, and `x * obj`, where **x** is **not an instance of the concerned class**.



Reflected Operators



Reflected arithmetic operators

<code>__radd__(self, other)</code>	<code>b+a</code>
<code>__rsub__(self, other)</code>	<code>b-a</code>
<code>__rmul__(self, other)</code>	<code>b*a</code>
<code>__rfloordiv__(self, other)</code>	<code>b // a</code>
<code>__rdiv__(self, other)</code>	<code>b / a</code>
<code>__rtruediv__(self, other)</code>	<code>b / a (from __future__ import division)</code>
<code>__rmod__(self, other)</code>	<code>b % a</code>
<code>__rdivmod__(self, other)</code>	<code>divmod(b, a)</code>
<code>__rpow__</code>	<code>b ** a</code>



Modified Example

```
class Book:  
    def __init__(self,name,price):  
        self.name=name  
        self.price=price  
    def __add__(self,other):  
        totalprice=self.price+other.price  
        return totalprice  
    def __radd__(self,other):  
        totalprice=self.price+other  
        return totalprice  
    def __str__(self):  
        return f'Name={self.name}, Price={self.price}'
```

```
b1=Book("Mastering Python",300)  
b2=Book("Mastering Java",500)  
b3=Book("Mastering C++",400)  
print(b1)  
print(b2)  
print(b3)  
print("Total price of books  
is:",b1+b2+b3)
```

Output:

```
name=Mastering Python, price=300  
name=Mastering Java, price=500  
name=Mastering C++, price=400  
Total price of books is: 1200
```

List Of Relational Operator For Overloading



Operator	Expression	Internally
Less than	$p1 < p2$	<code>p1.__lt__(p2)</code>
Less than or equal to	$p1 \leq p2$	<code>p1.__le__(p2)</code>
Equal to	$p1 == p2$	<code>p1.__eq__(p2)</code>
Not equal to	$p1 != p2$	<code>p1.__ne__(p2)</code>
Greater than	$p1 > p2$	<code>p1.__gt__(p2)</code>
Greater than or equal to	$p1 \geq p2$	<code>p1.__ge__(p2)</code>



Exercise



- Write a program to create a class called **Distance** having **2 instance members** called **feet** and **inches** . Provide following methods in **Distance** class
 - **__init__()** : This method should accept 2 arguments and initialize **feet** and **inches** with it
 - **__str__()**: This method should return string representation of **feet** and **inches**
 - **__eq__()** : This method should compare 2 **Distance** objects and return **True** if they are equal otherwise it should return **False**



Solution

```
class Distance:  
    def __init__(self,feet,inches):  
        self.feet=feet  
        self.inches=inches  
    def __eq__(self,other):  
        x=self.feet*12+self.inches  
        y=other.feet*12+other.inches  
        if x==y:  
            return True  
        else:  
            return False  
    def __str__(self):  
        return f'feet={self.feet},inches={self.inches}'
```

Output:

```
feet=0,inches=12  
feet=1,inches=0  
Distances are equal
```

```
d1=Distance(0,12)  
d2=Distance(1,0)  
print(d1)  
print(d2)  
if d1==d2 :  
    print("Distances are equal")  
else:  
    print("Distances are not equal")
```

List Of Shorthand Operator For Overloading



Operator	Expression	Internally
<code>-=</code>	<code>p1-=p2</code>	<code>p1.__isub__(p2)</code>
<code>+=</code>	<code>p1+=p2</code>	<code>p1.__iadd__(p2)</code>
<code>*=</code>	<code>p1*=p2</code>	<code>p1.__imul__(p2)</code>
<code>/=</code>	<code>p1/=p2</code>	<code>p1.__idiv__(p2)</code>
<code>//=</code>	<code>p1//=p2</code>	<code>p1.__ifloordiv__(p2)</code>
<code>%=</code>	<code>p1%=p2</code>	<code>p1.__imod__(p2)</code>
<code>**=</code>	<code>p1**=p2</code>	<code>p1.__ipow__(p2)</code>

List Of Special Functions/Operator: For Overloading



Function/Operator	Expression	Internally
len()	len(obj)	obj.__len__(self)
[]	obj[o]	obj.__getitem__(self,index)
in	var in obj	obj.__contains__(self,var)
str()	str(obj)	obj.__str__(self)

These Notes Have Python_world_In Notes Copy rights under copy right

Acts of Government of India. So commercial Use of this notes is Strictly prohibited



These Notes Have Python _ world _ In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 49



Today's Agenda

- **Database Programming In Python-I**
 - What Is Data And Database ?
 - What Is DBMS ?
 - What Is SQL ?
 - **How To Configure Our System For Database Programming In Python ?**



Introduction



- Before we learn about **Database Programming In Python**, let's first understand -
- **What is Data?**
- **What is Database ?**



Introduction



- ## What is Data?

- In simple words **data** can be **facts** or **information**.
- For example **your name**, **population** of a **country** , **names** of political parties in your **country** , **today's temperature** etc
- A **picture** , **image** , **file** , **pdf** etc can also be considered data.



Introduction



• What is a Database?

- A **database** is a **collection** of inter-related data or **information** that is organized so that it can easily be **accessed**, **managed**, and **updated**.
- Let's discuss few examples.
 - Your **mobile's phone book** is a **database** as it stores data pertaining to people like their **phone numbers**, **name** and **other contact details** etc.
 - Your **University** uses **database** to store **student details** like **enrollment no**, **name** , **address** , **academic performance** etc
 - Let's also consider the **Facebook**. It needs to store, manipulate and present data related to **members**, their **friends**, **member activities**, **messages**, **advertisements** and lot more. Here also **database** is used

How Databases Store The Data ?



- Most of the **databases** store their data in the form of **tables**
- Each **table** in a database has **one or more columns**, and each column is assigned a specific **data type**, such as an integer number, a sequence of characters (for text), or a date.
- Each **row** in the table has a value for each **column**.

How Databases Store The Data ?



Name	FName	City	Age	Salary
Smith	John	3	35	\$280
Doe	Jane	1	28	\$325
Brown	Scott	3	41	\$265
Howard	Shemp	4	48	\$359
Taylor	Tom	2	22	\$250



Components Of A Table



Column Names

Fields [columns]

Table

Key Field

agent_code	agent_name	working_area	commission	phone_no
A007	Ramasundar	Bangalore	0.15	077-25814763
A005	Anderson	Brisban	0.14	045-21447739
A001	Subbarao	Bangalore	0.14	077-12346674
A003	Alex	London	0.12	075-12458969
A008	Alford	New York	0.12	044-25874365

Record [Row]

Column Value

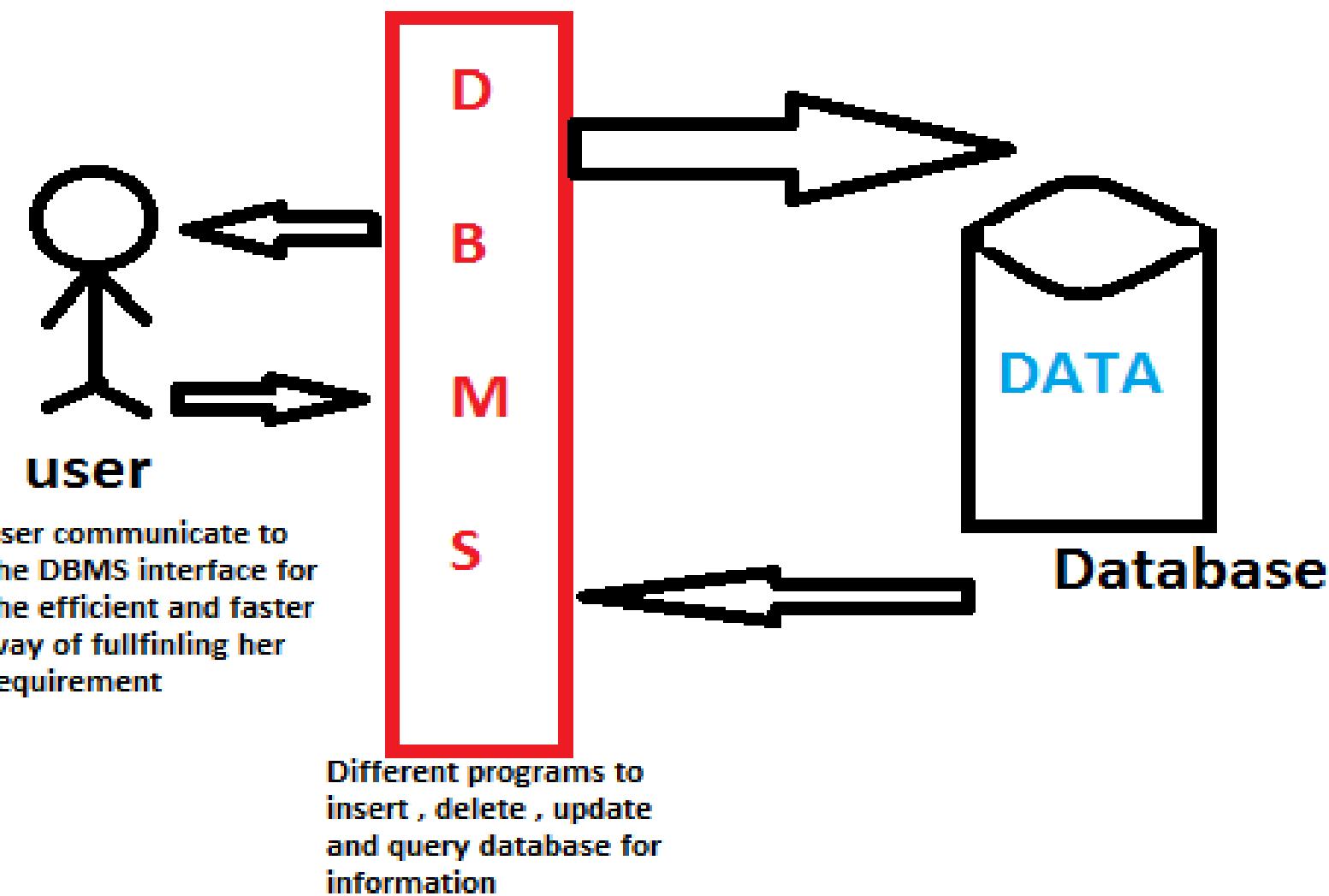


What Is A DBMS ?



- A **DBMS** is a program or a software that allows users to perform different **operations** on a database.
- These **operations** include:
 - **Creating the database/tables**
 - **Inserting records into these tables**
 - **Selecting records from these tables for displaying**
 - **Updating / Deleting the records**

What Is A DBMS ?





Some Popular DBMS

- Some of the most popular **DBMS** are:
 - **Oracle**
 - **MySQL**
 - **MS SQL Server**
 - **SQLite**
 - **PostgreSQL**
 - **IBM DB2**

and many more

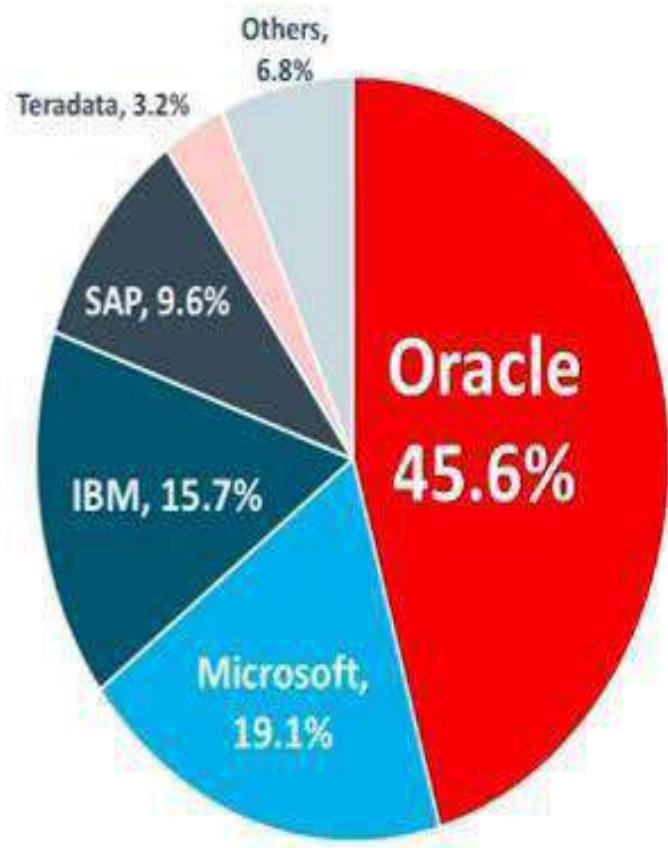
These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

The Market Leader



Oracle Continues Have Largest **RDBMS** Market Share by Wide Margin

**More than Double Sales of
Nearest Competitor**



Graphic created by Oracle based on Gartner



What Is SQL ?



- **SQL** is an abbreviation for “**Structured Query Language**”.
- It is a language used by **EVERY DBMS** to interact with the database.
- It provides us **COMMANDS** for **inserting data** to a database, **selecting data** from the database and **modifying data** in the database

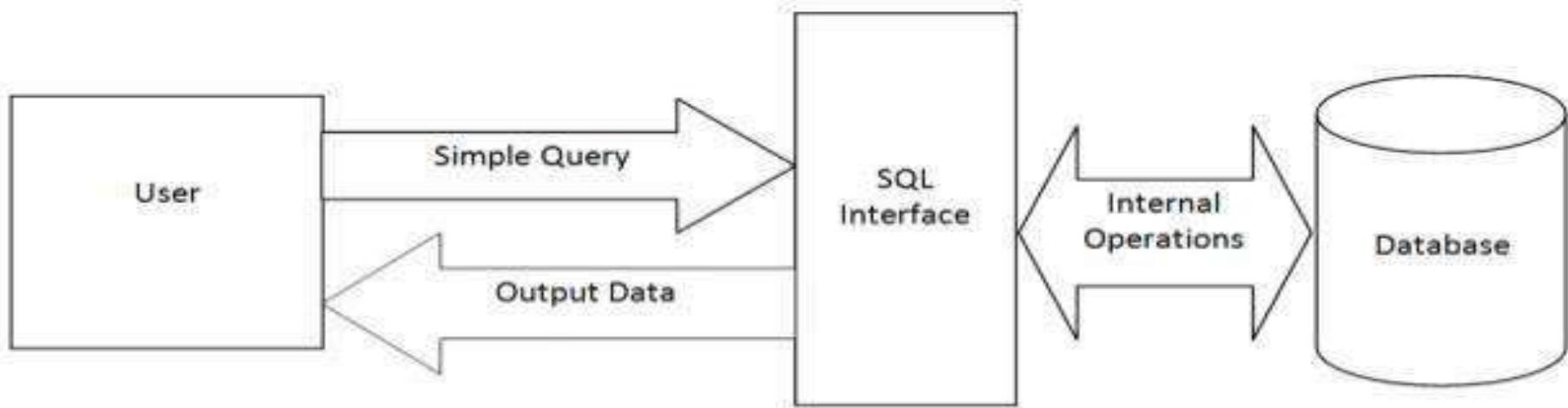


Pictorial View Of SQL



What is SQL?

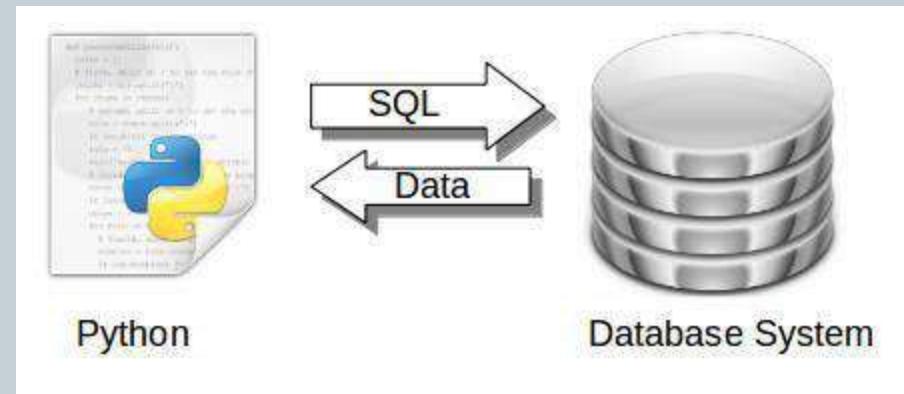
SQL is simply a language that makes it easy to pull data from your application's database.



Database Programming In Python



- **Python** is wonderfully able to interact with **databases**, and this is what we will learn in this chapter.
- **Advantages:**
 - Platform-independent
 - Faster and more efficient
 - Easy to migrate and port database application interfaces



How Python Connects To Database?



- Python uses the ***Python Database API*** in order to interact with databases.
- An **API** stands for **Application Programming Interface**.
- It is a **set** of **predefined functions** , **classes** and **methods** given by the **language** for a **particular task** and the programmer can use it whenever he wants to perform that task in his code.

How Python Connects To Database?



- The ***Python Database API*** allows us to handle **different database management systems** (DBMS) in our **Python code**.
- However the **steps at the code level remain altogether same**.
- That is using the same steps we can connect to **Oracle** or **MySQL** or **SQLite** or any other **DBMS**



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 50



Today's Agenda

- **Database Programming In Python-II**
 - Introduction To SQLite
 - Steps Needed For Connecting To SQLite From Python
 - Exploring Connection And Cursor Objects
 - Executing The SQL Queries
 - Different Ways Of Fetching The Data



Introduction To SQLite



- **SQLite** is an **Open Source Database** developed in **C** programming language.
- It stores data to a **text file** on the device and is **a popular choice** for application softwares such as web browsers as well as mobile platforms like **Android** and **iOS**.



SQLite Features



- **Extremely light-weighted** (not more than 500 KBs)
- **No complex setup**
- **Fully transactional.**
- It supports all standard relational database features like **SQL Queries**, **Joins** , **Constraints** etc



Who Uses Sqlite ?



Following are well known companies/products that use
SQLite:

Adobe

Apple

McAfee

Microsoft

DropBox

Facebook

Google

Python

PHP



SQLite Limitations



- **SQLite** supports neither **RIGHT OUTER JOIN** nor **FULL OUTER JOIN**. It supports only **LEFT OUTER JOIN**.
- With **ALTER TABLE** statement in **SQLite** we can only **add a column** or **rename a table** or **column**.
- However, we can't do the following:
 - **DROP a column.**
 - **ADD a constraint.**



SQLite Limitations



- **GRANT** and **REVOKE** commands are not implemented in **SQLite**.
- **VIEWS** are **read-only** – we can't write **INSERT**, **DELETE**, or **UPDATE** statements into the view.
- **SQLite** only supports **FOR EACH ROW** triggers, and it doesn't support **FOR EACH STATEMENT** triggers.



SQLite Installation

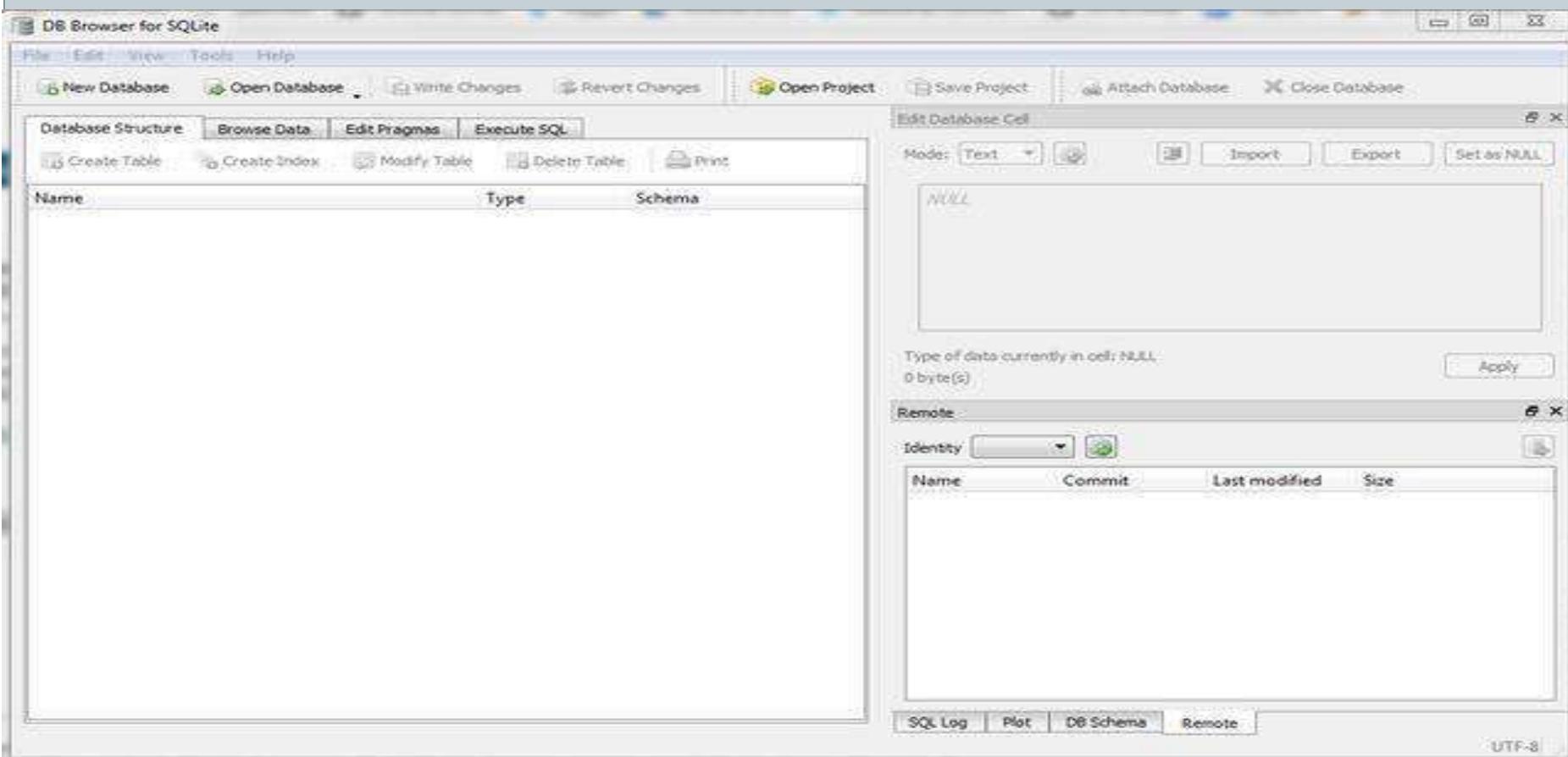


- **Installing** and **setting up SQLite** takes a matter of a few minutes.
- We can use **SQLite** from the command line tools, but there is a **GUI-based utility** which lets us use **SQLite** through a decent graphical interface.
- This is called **DB Browser for SQLite**.
- We can download it from
<https://sqlitebrowser.org/dl/> with respect to our OS platform.



SQLite Installation

- On Windows platform, the interface for DB Browser for SQLite looks like this –



Steps Required For Connecting Python Code To SQLite



- Connecting our **Python app** to any **database** involves total **6 important steps**.
- Also remember that ***these steps will always remain*** same ***irrespective of the database*** we are trying to connect
- The only ***difference*** will be the ***change*** in the ***name of the module***

Steps Required For Connecting Python Code To Sqlite



- For connecting to **SQLite** , the steps are:
 - Import the module **sqlite3**
 - Establish a **connection** to the database.
 - Create a **cursor** to communicate with the data.
 - **Execute** the SQL query
 - **Fetch** the result returned by the SQL query
 - **Close** the **cursor** and **connection** to the database.

Step 1- Importing The Module



- Since we are connecting to **SQLite**, so all the **functions** and **classes** we will be using will be supplied by the **module** called **sqlite3**.
- So the first step will be to **import** this **module** by writing the following statement:

```
import sqlite3
```

Step 2- Establishing The Connection



- After importing the module ,we must open the connection to the **SQlite**.
- This can be done by calling the function **connect()** of **sqlite3** module having the following syntax:

sqlite3.connect(“path to the db file”)

- Following is the description of this function:
 - It accepts a **connection string** as argument mentioning the path to the db file
 - If a connection is established, then a **Connection** object is returned.
 - If there is any problem in connecting to the database , then this function throws the exception called **sqlite3.DatabaseError**

Important Attributes/Methods Of Connection Object



- When the connection is successful, we get back the **sqlite3.Connection** object.
- This object provides us some important methods which are as follows:
 - **cursor()** : Return a new **Cursor** object using the **connection**.
 - **close()**: **Closes** the **connection**
 - **commit()**: **Commits** any pending **transactions** to the database.
 - **rollback()**: **Rollbacks** any pending **transactions**.

Step 3- Creating The Cursor



- Once we have a connection, we need to get a **cursor**
- A **Cursor** allows us to send all the **SQL commands** from our Python code to the database.
- It can also hold the set of rows returned by the query and lets us work with the records in that set, in sequence, one at a time.
- To get a **Cursor** object we call the method **cursor()** of the **Connection** object as follows:

```
cur=conn.cursor()
```

Important Attributes/Methods Of Cursor Object



- A **Cursor** object provides us some attributes and methods to execute the **SQL query** and get back the results
- Following are it's important **attributes**:
 - **rowcount**: Returns the number of rows fetched or affected by the last operation, or -1 if the module is unable to determine this value.
- Following are it's important **methods**:
 - **execute(statement)** : Executes an **SQL statement** string on the DB
 - **fetchall()**: Returns all remaining result rows from the last query as a sequence of tuples
 - **fetchone()**: Returns the next result row from the last query as a tuple
 - **fetchmany(n)**: Returns up to n remaining result rows from the last query as a sequence of tuples.
 - **close()**: Closes the cursor

Step 4- The **execute()** Method



- **Syntax:**

execute(SQL statement, [parameters], **kwargs)

- This method can accept an **SQL statement** - to be run directly against the database. It executes this SQL query and stores the result.
- **For example:**

cur.execute('select * from allbooks')



The `execute()` Method



- It can also accept **Bind variables** assigned through the **parameters** or **keyword arguments**.
- We will discuss this later

Step 5- Fetching The Result



- Once we have executed the **SELECT query** , we would like to retrieve the rows returned by it.
- There are numerous ways to do this:
 - By iterating directly over the **Cursor** object
 - By calling the method **fetchone()**
 - By calling the method **fetchall()**
- We will discuss each of these methods after **step 6**

Step 6- Fetching The Result



- The **final step** will be to **close the cursor** as well as **close the connection** to the database once we are done with processing.
- This is done by calling the method **close()** on both the objects.

- **Example:**

cur.close()

conn.close()



An Important Point!

- During communication with **Sqlite** , if any problem occurs the methods of the module **sqlite3** throw an exception called **DatabaseError**.
- So it is a best practice to execute **sqlite3** methods that access the database within a **try..except** structure in order to catch and report any exceptions that they might throw.

Directly Iterating Over The Cursor



- The **Cursor** object holds all the rows it retrieved from the database as **tuples**.
- So if we **iterate** over the **Cursor** object using the **for loop**, then we can retrieve these rows



Exercise



- Assume you have a table called **Allbooks** in the database which contains **4 columns** called **bookid**, **bookname**, **bookprice** and **subject**.
- Write a **Python** code to do the following:
 - Connect to the database
 - Execute the query to select **name of the book** and **it's price** from the table **Allbooks**
 - Display the records



Sample Output

```
D:\My Python Codes>python sqldbdemo1.py
Connected successfully to the DB
('Let Us C', 350)
('Learning Python', 400)
('Mastering HTML', 400)
('C In Depth', 300)
('Java Gems', 430)
('Let Us C++', 380)
('Projects In Java', 500)
Disconnected successfully from the DB
```



Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks")
    for x in cur:
        print(x)
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```



Exercise



- Modify the code so that values are displayed without **tuple symbol** i.e. without the symbol of ()
- Sample Output

```
D:\My Python Codes>python sqldbdemo2.py
Connected successfully to the DB
Let Us C 350
Learning Python 400
Mastering HTML 400
C In Depth 300
Java Gems 430
Let Us C++ 380
Projects In Java 500
Disconnected successfully from the DB
```



Solution

```
import sqlite3  
conn=None  
try:  
    conn=sqlite3.connect("d:/mysqitedb/library.db")  
    print("Connected successfully to the DB")  
    cur=conn.cursor()  
    cur.execute("Select bookname,bookprice from allbooks")  
    for name,price in cur:  
        print(name,price)  
except (sqlite3.DatabaseError)as ex:  
    print("Error in connecting to Sqlite:",ex)  
finally:  
    if conn is not None:  
        conn.close()  
        print("Disconnected successfully from the DB")
```

We will just have to **unpack** each row of the **tuple** to get the **individual values**



Using The Method **fetchone()**



- Sometimes we may want to pull just one record at a time from the table .
- As a result **Cursor** object provides us a method called **fetchone()** .
- This method returns **one record** as a **tuple**, and if there are no more records then it returns **None**



Exercise



- Modify the previous code to display the name and price of the **costliest** book from the table **Allbooks**



Sample Output

```
D:\My Python Codes>python sqldbdemo3.py
Connected successfully to the DB
('Projects In Java', 500)
Disconnected successfully from the DB
```



Solution



```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks order by bookprice desc")
    x=cur.fetchone()
    if x is not None:
        print(x)

except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```



Using The Method **fetchall()**



- **Syntax:**

cur.fetchall()

- The method fetches all rows of a query result set and returns it as a **list** of **tuples**.
- If no more rows are available, it returns an **empty list**.



Sample Code

```
import cx_Oracle
conn=None
cur=None
try:
    conn=cx_Oracle.connect("scott/tiger@Sachin-PC/orcl")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks order by bookprice desc")
    x=cur.fetchall()
    print(x)
except(cx_Oracle.DatabaseError)as ex:
    print("Error in connecting to Oracle:",ex)
finally:
    if cur is not None:
        cur.close()
        print("Cursor closed successfully")
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```



Output

Connected successfully to the DB

```
[('Projects In Java', 500), ('Java Gems', 430), ('Learning Python', 400), ('Mastering HTML', 400), ('Let Us C++', 380), ('Let Us C', 350), ('C In Depth', 300)]
```

Disconnected successfully from the DB



Exercise



- Modify the previous book application so that your code asks the user to enter a record number and displays only that book .
- Make sure if the record number entered by the user is wrong then display appropriate error message



Output



```
Connected successfully to the DB
Enter the record number(1 to 7):2
Java Gems 430
Disconnected successfully from the DB
```

```
D:\My Python Codes>python sqldbdemo5.py
Connected successfully to the DB
Enter the record number(1 to 7):6
Let Us C 350
Disconnected successfully from the DB
```

```
D:\My Python Codes>python sqldbdemo5.py
Connected successfully to the DB
Enter the record number(1 to 7):12
Record number should be between 1 to 7
Disconnected successfully from the DB
```



Solution

```
import sqlite3  
conn=None  
try:  
    conn=sqlite3.connect("d:/mysqitedb/library.db")  
    print("Connected successfully to the DB")  
    cur=conn.cursor()  
    cur.execute("Select bookname,bookprice from allbooks  
    order by bookprice desc")  
    booklist=cur.fetchall()  
    recnum=int(input("Enter the record number(1 to  
    "+str(len(booklist))+":"))
```



Solution

```
if recnum <1 or recnum>len(booklist):
    print("Record number should be between 1 to
"+str(len(booklist)))
else:
    row=booklist[recnum-1]
    print(row[0],row[1])

except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

LECTURE 51



Today's Agenda

- **Database Programming In Python-III**
 - Executing INSERT Command
 - Executing Dynamic Queries
 - Concept Of Bind Variables
 - Executing Update Command
 - Executing Delete Command



Inserting Record



- To insert a new record in the table we have to execute the **INSERT INTO** command.
- It has **2 syntaxes**:
 - **Insert into <table_name> values(<list of values>)**
 - **Insert into <table_name>(<list of cols>) values(<list of values>)**



Inserting Record



- To insert a record in the table from our Python code we simply pass the **insert query** as argument to the **execute()** method of **cursor** object.
- It's general syntax is:
cur.execute("insert query")
- **Two important points:**
 - After executing insert if we want to get the number of row inserted we can use the **cursor** attribute **rowcount**
 - Unless we call the method **commit()** of **connection** object , the record we insert does not get saved in the table

Steps Required For Inserting Records



- For inserting record the overall steps are:
 - Import the module **sqlite3**
 - Establish a **connection** to the database.
 - Create a **cursor** to communicate with the data.
 - **Execute** the **Insert** query
 - **Commit** the changes
 - **Close** the connection to the database.



Example

```
import sqlite3  
conn=None  
try:
```

```
    conn=sqlite3.connect("d:/mysqitedb/library.db")  
    print("Connected successfully to the DB")  
    cur=conn.cursor()  
    cur.execute("Insert into allbooks values(108,'Python Web  
    Prog',500,'Python')")  
    n=cur.rowcount  
    print(n," row inserted")  
    conn.commit()
```

```
except(sqlite3.DatabaseError)as ex:  
    print("Error in connecting to Sqlite:",ex)  
finally:
```

```
    if conn is not None:  
        conn.close()  
        print("Disconnected successfully from the DB")
```

```
Connected successfully to the DB  
1 row inserted  
Disconnected successfully from the DB
```



Executing Dynamic Queries

- **Dynamic queries** are those where we **set the values** to be **passed in the query** at **run time**.
- For example , we would like to **accept the values** of the **record to be inserted from the user** and then pass it to the insert query.
- Such queries are called **dynamic query**



Executing Dynamic Queries

- **Dynamic queries** for **Sqlite** can be set using the concept of **bind variables**
- **Bind variables** are like **placeholders** used in a query , represented using **:some_number**, and are replaced with actual values before query execution.

Using bind variables

By Position



```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqldb/library")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    id=int(input("Enter bookid:"))
    name=input("Enter bookname:")
    price=int(input("Enter bookprice:"))
    subject=input("Enter subject:")
    cur.execute("Insert into allbooks values(:1,:2,:3,:4)",(id,name,price,subject))
    n=cur.rowcount
    print(n, " row inserted")
    conn.commit()
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```

In this Insert query **:1**,**:2**,**:3** and **:4** are called bind variables and they will be replaced with the values of the actual variables **id**,**name**,**price** and **subject** before the query is sent for execution.
Also , parenthesis is required because these values are sent as a tuple



Sample Output

```
Connected successfully to the DB
Enter bookid:130
Enter bookname:PHP 7
Enter subject:PHP
Enter bookprice:450
1 row inserted
Cursor closed successfully
Disconnected successfully from the DB
```



Updating Record

- To update a record in the table we have to execute the **UPDATE** command.
- It has **2 syntaxes**:
 - **Update <table name> set <col name>=<value>**
 - **Update <table name> set <col name>=<value> where <test condition>**



Updating Record

- Updating a record through **Python code** is same as inserting a new record..
- We call the method **execute()** of **cursor** object passing it the **update query**.
- It's **general syntax** is:
cur.execute("update query")
- If the query is dynamic then we can use **bind variables** for setting the values at run time.



Example

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqldatabase/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Update allbooks set bookprice=500 where bookid=201 ")
    n=cur.rowcount
    print(n," row updated")
    conn.commit()
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```



Exercise



- Write a program to accept a **subject name** and an **amount** from the user and **increase the price** of all the books of the **given subject** by adding the amount in the **current price**. Finally display whether books were updated or not and how many books were updated



Sample Output

Sample Run 1

```
Connected successfully to the DB
Enter subject name:Python
Enter the amount to increase:200
2 rows updated
Cursor closed successfully
Disconnected successfully from the DB
```

Sample Run 2

```
Connected successfully to the DB
Enter subject name:RoR
Enter the amount to increase:200
No rows updated
Cursor closed successfully
Disconnected successfully from the DB
```



Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqitedb/library.db")
    print("Connected successfully to the DB")
    subject=input("Enter subject name:")
    amount=int(input("Enter the amount to increase:"))
    cur=conn.cursor()
    cur.execute("Update allbooks set bookprice=bookprice+:1 where
    subject=:2",(amount,subject))
    n=cur.rowcount
    if n==0:
        print("No rows updated")
    else:
        print(n, "rows updated")
        conn.commit()

except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```



Deleting Record

- To delete a record from the table we have to execute the **DELETE** command.
- It has **2 syntaxes:**
 - **Delete from <table name>**
 - **Delete from <table name> where <test condition>**



Deleting Record

- Deleting a record through **Python code** is same as updating/inserting a record. .
- We call the method **execute()** of **cursor** object passing it the **delete query**.
- It's **general syntax** is:
cur.execute("delete query")
- If the query is dynamic then we can use **bind variables** for setting the values at run time.



Example

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Delete from allbooks where bookid=109")
    n=cur.rowcount
    print(n, " row deleted")
    conn.commit()

except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```



Exercise



- Write a program to accept a **subject name** from the user and **delete** all the books of the **given subject**. Finally display whether books were deleted or not and how many books were deleted



Sample Output

Sample Run 1

```
Connected successfully to the DB
Enter subject name:JS
1 rows deleted
Cursor closed successfully
Disconnected successfully from the DB
```

Sample Run 2

```
Connected successfully to the DB
Enter subject name:JS
No rows deleted
Cursor closed successfully
Disconnected successfully from the DB
```



Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    subject=input("Enter subject name:")
    cur.execute("Delete from allbooks where subject=:1",(subject,))
    n=cur.rowcount
    if n==0:
        print("No rows deleted")
    else:
        print(n, " rows deleted")
        conn.commit()
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
    if conn is not None:
        conn.close()
        print("Disconnected successfully from the DB")
```

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



PYTHON

LECTURE 52



Today's Agenda

• **File Handling**

- What Is File Handling ?
- What Is The Need Of File Handling ?
- Examples Where Files Are Used?
- Python's Way Of Handling Files
- File Opening Modes
- Writing In A File
- Different Ways For Reading From A File



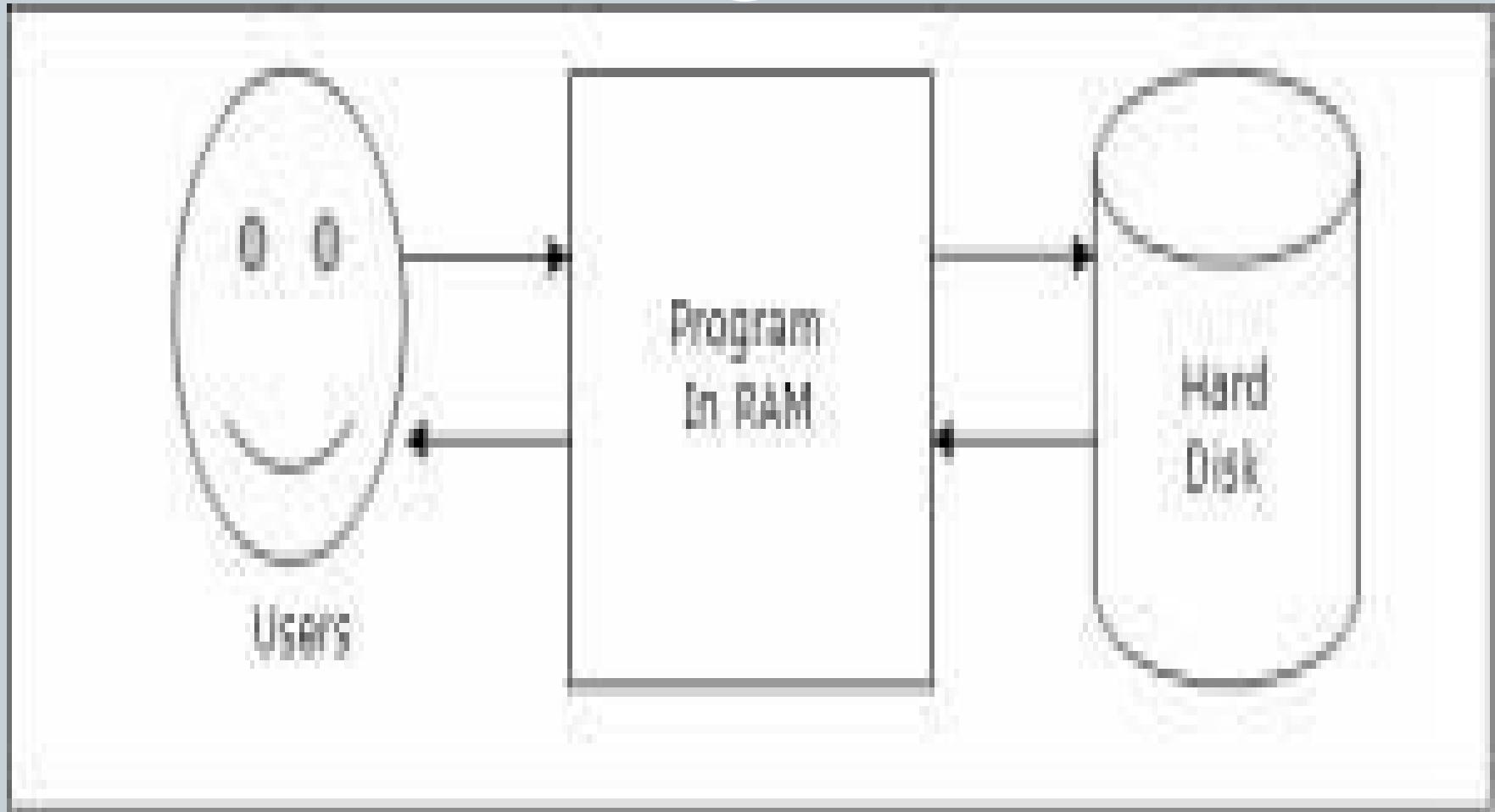
What Is File Handling ?



- **File handling** is the process of accessing data files stored in the **secondary memory** of our computer.
- It allows us to **perform various operations** on these files **through our program** like **renaming files**, **deleting file**, **moving file** and above all **reading** and **writing** the contents in a File



What Is File Handling ?



Real Life Examples Of File Handling



- **Mobile's Phonebook**
- **Computer/Mobile Games**
- **Call Logs**
- **Gallery In Mobile**
- **User Accounts In Operating System**
- **Windows Registry**



Steps Needed For File Handling



- Broadly , file handling involves **3 steps:**
 - **Open the file.**
 - **Process file i.e perform read or write operation.**
 - **Close the file.**



Step -1 : Opening The File



- Before we can perform any operation on a file, we must open it.
- **Python** provides a **function** called **open()** to open a file.
- **Syntax:**

fileobject = open(filename, mode)

- The **filename** is the name or path of the file.
- The **mode** is a string which specifies the type operation we want to perform on the file (i.e **read**, **write**, **append**, etc). Default is **read**



File Opening Modes

Mode	Description
"r"	Opens the file for reading. If the file doesn't already exists we will get FileNotFoundException exception
"w"	Opens the file for writing. In this mode, if file specified doesn't exists, it will be created. If the file exists, then it's data is destroyed. If the path is incorrect then we will get FileNotFoundException exception
"a"	Opens the file in append mode. If the file doesn't exists this mode will create the file. If the file already exists then it appends new data to the end of the file rather than destroying data as "w" mode does.



File Opening Modes

Mode	Description
“r+”	Opens file for both reading and writing.
“w+”	Opens a file for both writing and reading. If the file exists then it will overwrite it otherwise it will create it.
“a+”	Opens file for appending and reading. If the file already exists then pointer will be set at the end of the file otherwise a new file will be created.



Examples Of Opening File

- **Example 1:**
 - `f = open("employees.txt", "r")`
- This statement opens the file **employees.txt** for **reading**.

- **Example 2:**
 - `f = open("teams.txt", "w")`
- This statement opens the file **teams.txt** in **write mode**.

- **Example 3:**
 - `f = open("teams.txt", "a")`
- This statement opens the file **teams.txt** in **append mode**.



Examples Of Opening File



- Instead of using **relative file paths** we can also use **absolute file paths**.
- **For example:**
 - `f = open("C:/Users/sachin/documents/README.txt", "w")`
- This statement opens the text file **README.txt** that is in **C:\Users\sachin\documents** directory in **write mode**.



Examples Of Opening File



- We can also use something called "**raw string**" by specifying **r** character in front of the **string** as follows:
 - `f = open(r"C:\Users\sachin\documents\README.txt", "w")`
- The **r** character causes the **Python** to treat every character in string as literal characters.



Step -3 : Closing The File



- Once we are done working with the file or we want to open the file in some other mode, we should close the file using **close()** method of the file object as follows:
 - **f.close()**



The **TextIOWrapper** Class



- The file object returned by **open()** function is an object of type **TextIOWrapper**.
- The class **TextIOWrapper** provides methods and attributes which helps us to **read** or **write** data from and to the file.
- In the next slide we have some commonly used methods of **TextIOWrapper** class.

Methods Of The **TextIOWrapper** Class



Method	Description
read([num])	Reads the specified number of characters from the file and returns them as string. If num is omitted then it reads the entire file.
readline()	Reads a single line and returns it as a string.
readlines()	Reads the content of a file line by line and returns them as a list of strings.
write(str)	Writes the string argument to the file and returns the number of characters written to the file.
seek(offset, origin)	Moves the file pointer to the given offset from the origin.
tell()	Returns the current position of the file pointer.
close()	Closes the file

Exceptions Raised In File Handling



- **Python** generates **many exceptions** when something goes wrong while interacting with files.
- The 2 most common of them are:
 - **FileNotFoundException**: Raised when we try to open a file that doesn't exist
 - **OSError**: Raise when an operation on file cause system related error.



Exercise



- Write a program to create a file called **message.txt** in **d:** of your computer .
- Now ask the user to **type a message** and **write it** in the file .
- Finally display **how many capital letters**, **how many small letters** , **how many digits** and **how many special characters** were written in the file.
- Also properly handle every possible exception the code can throw



Sample Output

```
Type a message:Happy New Year , 2019
File saved successfully!
Total upper case letters are : 3
Total lower case letters are : 9
Total digits are : 4
Total special characters are : 5
File closed successfully
```



Solution

```
fout=None
try:
    fout=open("d:\\message.txt","w")
    text=input("Type a message:")
    upper=0
    lower=0
    digits=0
    for ch in text:
        fout.write(ch)
        if 65<=ord(ch)<=90:
            upper+=1
        elif 97<=ord(ch)<=122:
            lower+=1
        elif 48 <=ord(ch)<=57:
            digits+=1
```

```
print("File saved successfully!")
print("Total upper case letters are :",upper)
print("Total lower case letters are :",lower)
print("Total digits are :",digits)
print("Total special characters are :",
      len(text)-(lower+upper+digits))
except FileNotFoundError as ex:
    print("Could not create the file: ",ex)

except OSError:
    print("Some error occurred while writing")

finally:
    if not fout is None:
        fout.close()
        print("File closed successfully")
```



Exercise



- Write a program to open the **message.txt** created by the previous code.
- Now read and display the contents of the file.
Click to add text
- Also properly handle every possible exception the code can throw



Solution

```
fin=None  
try:  
    fin=open("d:\\message.txt","r")  
    text=fin.read()  
    print(text)  
except FileNotFoundError as ex:  
    print("Could not open the file: ",ex)  
  
finally:  
    if fin is not None:  
        fin.close()  
        print("File closed successfully")
```

Happy New Year , 2019
File closed successfully



Exercise



- Write a program to create a file called **message.txt** in **d:** of your computer .
- Now ask the user to continuously type messages and save them in the file line by line.
- Stop when the user strikes an **ENTER** key on a **new line**
- Finally display **how many lines** were written in the file.
- Also properly handle every possible exception the code can throw



Sample Output

Type your message and to stop just press ENTER on a newline
Hello Everyone,
Wish you all a very happy and prosperous new year.
May you get whatever you deserve.

File saved successfully!
Total lines written are : 3
File closed successfully



Solution

```
fout=None
try:
    fout=open("d:\\message.txt","w")
    text=input("Type your message and to stop just press ENTER on a newline\\n")
    lines=0
    while True:
        if text=="":
            break
        lines+=1
        fout.write(text+"\n")
        text=input()

    print("File saved successfully!")
    print("Total lines written are :",lines)

except FileNotFoundError as ex1:
    print("Could not create the file: ",ex1)

except OSError as ex2:
    print(ex2)

finally:
    if fout is not None:
        fout.close()
        print("File closed successfully")
```

Type your message and to stop just press ENTER on a newline
Hello Everyone,
Wish you all a very happy and prosperous new year.
May you get whatever you deserve.

File saved successfully!
Total lines written are : 3
File closed successfully



Exercise



- Write a program to open the **message.txt** created by the previous code.
- Now **read** and **display** the contents of the **file** line by line.
- Finally also display **total number of lines** read from the file.
- Also properly handle every possible exception the code can throw



Sample Output

Hello Everyone,
Wish you all a very happy and prosperous new year.
May you get whatever you deserve.
Total lines read are : 3
File closed successfully



Solution

```
try:  
    fin=open("d:\\message.txt","r")  
    lines=0  
    while True:  
        text=fin.readline()  
        if text=="":  
            break  
        lines+=1  
        print(text,end="")  
  
    print("Total lines read are :",lines)  
  
except FileNotFoundError as ex:  
    print("Could not open the file: ",ex)  
  
finally:  
    if fin is not None:  
        fin.close()  
        print("File closed successfully")
```

```
Hello Everyone,  
wish you all a very happy and prosperous new year.  
May you get whatever you deserve.  
Total lines read are : 3  
File closed successfully
```

Using for Loop To Read The File



- Python allows us to use **for loop** also to read the contents of the **file line by line**.
- This is because the object of **TextIOWrapper** is also a kind of **collection/sequence** of characters fetched from the file.
- The only point is that when we use **for loop** on the **file object**, Python reads and returns **one line at a time**.



Exercise



- Write a program to open the **message.txt** created by the previous code.
- Now **read** and **display** the contents of the **file** line by line.
- Finally also display **total number of lines** read from the file.
- Also properly handle every possible exception the code can throw



Solution



```
fin=None
try:
    fin=open("d:\\message.txt","r")
    lines=0
    for text in fin:
        print(text,end="")
        lines+=1
    print("Total lines read are :",lines)

except FileNotFoundError as ex:
    print("Could not open the file: ",ex)

finally:
    if fin is not None:
        fin.close()
        print("File closed successfully")
```

```
Hello Everyone,
wish you all a very happy and prosperous new year.
May you get whatever you deserve.
Total lines read are : 3
File closed successfully
```

These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



PYTHON

LECTURE 53



Today's Agenda



- **File Handling**
 - The seek() Method
 - Appending In A File
 - Using with Statement



The **seek()** Method

- To reset the internal file pointer we use the **seek()** method.
- **Syntax:**
seek(offset, whence).
- **whence** is optional, and determines where to seek from.
 - If whence is 0, the bytes/letters are counted from the beginning.
 - If it is 1, the bytes are counted from the current cursor position.
 - If it is 2, then the bytes are counted from the end of the file.
 - If nothing is put there, 0 is assumed.
- **offset** describes how far from whence that the cursor moves.



seek() Examples



- **f.seek(45,0)**
 - would move the cursor to **45** bytes/letters after the **beginning** of the file.
- **f.seek(10,1)**
 - would move the cursor to **10** bytes/letters after the **current** cursor position.
- **f.seek(-77,2)**
 - would move the cursor to **77** bytes/letters before the **end** of the file (notice the **-** before the **77**)
- Special Note:
 - Python **3** only supports **text file** seeks from the beginning of the file.
 - If we try to call **seek()** with **non-zero value** for offset from **cur** or **end** then the code will throw an exception called:
 - **UnsupportedOperation: can't do nonzero end-relative seeks**
 - Thus for **seek()** to work , the file must be a **binary file like** image file, music file etc.



Exercise



- Write a program to create the file **message.txt** in **d:** of your computer
- Now ask the user to continuously type messages and save them in the file line by line.
- Stop when the user strikes an ENTER key on a new line
- Finally display **how many lines** were written in the file.
- Now read and display the contents of the file line by line.
- Finally also display total number of lines read from the file.
- Also properly handle every possible exception the code can throw



Sample Output

Type your message and to stop just press ENTER on a newline
Sharma Computer Academy,
Pb-5, C-Block,
Mansarovar Complex,
Bhopal

File saved successfully!
Total lines written are : 4
Press any key to read the data:

Sharma Computer Academy,
Pb-5, C-Block,
Mansarovar Complex,
Bhopal

Total lines read are : 4
File closed successfully



Solution

try:

```
fobj=open("d:\\message.txt","w+")
text=input("Type your message and to stop just press ENTER on a newline\\n")
lines=0
```

while True:

```
    if text=="":
        break
    lines+=1
    fobj.write(text+"\n")
    text=input()
```

```
print("File saved successfully!")
print("Total lines written are :",lines)
print("Press any key to read the data:")
input()
fobj.seek(0)
lines=0
```

Solution



while True:

```
    text=fobj.readline()
    if text=="":
        break
    lines+=1
    print(text,end="")
```

```
print("Total lines read are :",lines)
```

except FileNotFoundError as ex1:

```
    print("Could not open the file: ",ex1)
```

except OSError as ex2:

```
    print("Error in file I/O: ",ex2)
```

finally:

```
    if 'fobj' in globals():
        fobj.close()
    print("File closed successfully")
```

These Notes Have Python _ world _ In Notes Copyrights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited





Appending Data In a File



- We can use "**a**" mode to **append** data to end of the **file**.
- When we open the file in "**a**" or "**a+**" mode , the internal **file pointer** is placed **at the end** while **writing** the new data.
- Thus , the new text we write **does not overwrite** the previous contents of the file.
- Moreover if the file is **not present** it gets **created**.



Example

```
days=["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]
fobj=None
try:
    fobj=open("d:\\days.txt","a")
    items=0
    for day in days[:3]:
        fobj.write(day+"\n")
        items+=1
    fobj.close()
    print(items,"values written")
    print("File closed")
    items=0
    fobj=open("d:\\days.txt","a")
    for day in days[3:]:
        fobj.write(day+"\n")
        items+=1
    fobj.close()
    print(items,"more values written")
    print("File closed")
    print("Press any key to read back the file")
    input()
```

```
3 values written
File closed
4 more values written
File closed
Press any key to read back the file
```



Example

```
fobj=open("d:\\days.txt","r")
for day in fobj:
    print(day,end="")

except FileNotFoundError as ex:
    print("Could not open the file: ",ex)

finally:
    if fobj is not None:
        fobj.close()
        print("File closed successfully")
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
File closed successfully
```



Using **with** Statement



- **Python** also provides a nice shortcut for **file handling** using the **with** statement.
- The following is the general form of the **with** statement when used with files.

with open(filename, mode) as file_object:

```
# body of with statement  
# perform the file operations here
```

Important Points About **with** Statement



- It automatically closes the file without requiring any work on our part.
- The scope of **file_object** variable is only limited to the body of the with statement.
- So , if we try to call **read()** or **write()** method on it outside the block we will get **NameError**.



Example

try:

```
with open("d:\\message.txt","r") as fin:
```

```
    lines=0
```

```
    for x in fin:
```

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

```
        lines+=1
```

```
    print("Total lines read are :",lines)
```

```
    print("File closed successfully!")
```

```
except FileNotFoundError as ex:
```

```
    print("Could not open the file: ",ex)
```



These Notes Have Python_world_In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



PYTHON

LECTURE 54



Today's Agenda



• **Modules**

- What is a module ?
- Different ways of importing module
- The concept of the variable `__name__`



What Is A **Module** ?



- Python **modules** are **.py** files that consist of **Python code**.
- So all the **Python programs** that we have written can be called a **module**.

From Where These Modules Come ?



- Modules are available in **3 ways**:
 - Some modules are available through the **Python Standard Library** and are therefore installed with our Python installation. Examples of such modules are: **math, sys,random** etc
 - We also can install modules developed by other programmers using **Python's package manager** called **pip**.
 - Additionally, we can create **our own Python modules** since modules are comprised of Python **.py** files.



What A **Module** Can Contain ?



- A **Python** module can contain any valid **Python** element like:
 - **variables**
 - **functions**
 - **classes**



Defining A Module

```
def greet(name):  
    print("Good Morning ",name,"!")
```

- Save the above code in a file called **welcome.py**

welcome.py

```
def greet(name):  
    print("Good Morning ",name,"!")
```



Using A Module



- To use a module we must **import** it in our program which can be done in **4** ways:
 - Using import
 - Using aliasing
 - Using from
 - Using wildcard

First Way (Direct import)



- Create another **Python** file called **test.py** and import the module **welcome.py** in it.
- Now we can call the function **greet()** by using the syntax as shown below:
- **Syntax:**

welcome.greet(<arg>)

First Way (Direct import)



test.py

```
import welcome
name=input("What is your name?")
welcome.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

Good Morning Sachin !

Second Way (Using aliasing)



- In the previous example we had to prefix the name of the module called **welcome** before the name of it's function **greet()**
- In order to shorten this syntax , **Python** allows us to **alias** the module name using the keyword **as**
- **Syntax:**
`import <module_name> as <new_name>`

Second Way (Using aliasing)



test.py

```
import welcome as w
name=input("What is your name?")
w.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

Good Morning Sachin !



Guess The Output ?



test.py

```
import welcome as w
name=input("What is your name?")
welcome.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

NameError: name 'welcome' is not defined



Guess The Output ?

test.py

```
import welcome
import welcome as w
name=input("What is your name?")
welcome.greet(name)
w.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin
Good Morning Sachin!
Good Morning Sachin!

These Notes Have Python _ world _ In Notes Copy rights under copy right
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



Third Way



- If we do not want to prefix the module name at all with any prefix then we must **import** specific members of a **module**
- To do this , **Python** provides us **from keyword**
- **Syntax:**
 - **from modname import name1[, name2[, ... nameN]]**



Third Way



test.py

```
from welcome import greet  
name=input("What is your name?")  
greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

Good Morning Sachin !



Guess The Output ?



m1.py

```
def greet(name):  
    print("Good Morning ",name,"!")
```

m2.py

```
def greet(name):  
    print("Good Evening ",name,"!")
```

test.py

```
from m1 import greet  
from m2 import greet  
name=input("What is your name?")  
greet(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin
Good Evening Sachin !



Guess The Output ?

m1.py

```
def greet(name):
    print("Good Morning ",name,"!")
```

m2.py

```
def greet(name):
    print("Good Evening ",name,"!")
```

test.py

```
from m1 import greet
from m2 import greet
name=input("What is your name?")
m1.greet(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin
NameError: name 'm1' is not defined



Guess The Output ?



welcome.py

```
def greet1(name):
    print("Good Morning ",name,"!")

def greet2(name):
    print("Good Afternoon ", name, "!"")
```

test.py

```
from welcome import greet1
name=input("What is your name?")
greet1(name)
greet2(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin
Good Morning Sachin !

NameError: name 'greet2' is not defined



Fourth Way



- It is also possible to **import all names** from a **module** into the current file by using the **wildcard character ***
- **Syntax:**
 - `from modname import *`
- This allows us to use all the items from a **module** into the **current file**



Fourth Way

welcome.py

```
def greet1(name):
    print("Good Morning ",name,"!")

def greet2(name):
    print("Good Afternoon ", name, "!"")
```

test.py

```
from welcome import *
name=input("What is your name?")
greet1(name)
greet2(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin
Good Morning Sachin !
Good Afternoon Sachin !

Defining A Class In A Module



- It is also possible to define a **class** inside a **module** containing **attributes** and **methods**
- It can be accessed in the same way like we can access functions

Defining A Class In A Module



shape.py

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        print("Area is", math.pi * math.pow(self.radius, 2))
```

Using The Class Outside The Module



useshape.py

```
import shape
radius=int(input("Enter radius:"))
obj=shape.Circle(radius)
obj.area()
```

Run:

python useshape.py

Output:

Enter radius:3

Area is 28.274333882308138

The Concept Of - **name==‘main’**



- Now after learning basics of modules , let us discuss another very important concept in **Python**.
- All programmers who write **standard code** in **Python** always include a test condition in their **module** which is something like :

```
if_  name==‘ main_ ’:  
    Some code
```

- So let us understand what it is all about

The Concept Of name____=='main__'



- Before we start the discussion , guess the output of the following code:

calculate.py

```
def add(a, b):
    print("Sum of",a,"and",b,"is",a+b)

def subtract(a, b):
    print("Diff of", a, "and", b, "is", a - b)
```

Run:

python calculate.py

Output:

Why did the code not produce any output?

This is because **add()** and **subtract()** are functions and functions are executed only when they are called .

But since we didn't call any of these functions so no output was produced

The Concept Of

- **name__==‘main__’**



- Now , guess the output:

calculate.py

```
def add(a, b):  
    print("Sum of",a,"and",b,"is",a+b)  
  
def subtract(a, b):  
    print("Diff of", a, "and", b, "is", a - b)  
  
add(10,20)  
subtract(10,20)
```

Run:

python calculate.py

Output:

Sum of 10 and 20 is 30
Diff of 10 and 20 is -10

As expected the code is showing the results of both **add()** and **subtract()** functions as we have called both of them

The Concept Of name__==‘ main_ ’



- Now suppose we **import** the previous **calculate** module in another file and call only the **add()** function.
- Can you guess the output then ?**



Guess The Output

calculate.py

```
def add(a, b):  
    print("Sum of",a,"and",b,"is",a+b)  
  
def subtract(a, b):  
    print("Diff of", a, "and", b, "is", a - b)  
  
add(10, 20)  
subtract(10, 20)
```

Execution

Run:

python test.py

Output:

Sum of 10 and 20 is 30

Diff of 10 and 20 is -10

Sum of 5 and 7 is 12

test.py

```
import calculate  
calculate.add(5,7)
```



Why 3 Outputs ?

- Whenever we import a **module** , Python immediately executes it.
- So as soon as **Python** found the statement:
import calculate
- It imported the module **calculate** and executed all the **global statements** in it.
- And since our module **calculate** contained 2 function calls **add(10,20)** and **subtract(10,20)** so the overall output contains outputs of these calls also

How To Overcome This Problem ?



- This is where the concept of a special variable called **name** comes into picture.
- For every module , **Python** creates a variable called **name**, which contains the **name of the module**.
- But the point to understand is that:
 - When we run a module as a **stand alone file** then this variable has the value “**_ main_**”
 - Otherwise this variable has the **name of the file** as its value

The Concept Of name__==‘main__’



- To understand the concept **look carefully** at the output of the code:

demo.py

```
print("demo.py module name is:", __name__)
```

Run:

```
python demo.py
```

Output:

```
demo.py module name is: __main__
```

Since we are running the file **demo.py** as a stand alone module so the variable **__name__** has been set to the value “**__main__**”

The Concept Of name__ == ' main__'



- Now consider the code below and it's output:

demo.py

```
print("demo.py module name is:", __name__)
```

sample.py

```
import demo
```

Run:

```
python sample.py
```

Output:

```
demo.py module name is: demo
```

Now the output is different because we are importing the module **demo** in another module.

As mentioned previously in this case the value of the variable **__name__** is the name of the file so the output is **demo**

The Concept Of - **name__==‘main_’**



- Now , we can solve our problem , so that if we run the module **calculate** as a stand alone module then both the functions , **add()** and **subtract()** , should run and if we run it as a part of another module then only that function should run which we have called.



Guess The Output

calculate.py

```
def add(a, b):  
    print("Sum of",a,"and",b,"is",a+b)  
  
def subtract(a, b):  
    print("Diff of", a, "and", b, "is", a - b)  
  
if __name__=="__main__":  
    add(10, 20)  
    subtract(10, 20)
```

Execution

Run:

python test.py

Output:

Sum of 5 and 7 is 12

test.py

```
import calculate  
calculate.add(5,7)
```

These Notes Have Python_world_In Notes Copy rights under copy right

Acts of Government of India. So commercial Use of this notes is Strictly prohibited

CONTENT

Chapter 1: Introduction To C

Chapter 2: Start Learning C

Chapter 3: Arithmetic Instruction

Chapter 4: Control Instruction

Chapter 5: Iterative Control Instruction

Chapter 6: Function and Storage Classes

Chapter 7: Arrays

Chapter 8: Strings

Chapter 9: Pointers

Chapter 10: Structure and DMA

Chapter 1: Introduction To C

Content

- 1.1 What is a Computer?
- 1.2 Does computer really work on 0 and 1
- 1.3 Hardware
- 1.4 Files
- 1.5 Program and process
- 1.6 Software and their types
- 1.7 About Operating System
- 1.8 How a program executes?
- 1.9 How to develop software in C?
- 1.10 History of C
- 1.11 IDE
- 1.12 Technical Terms
- 1.13 First C Program
- 1.14 Understanding Program Structure
- 1.15 Execution of C Program

1.1 What is a Computer?

Computer is an electronic device that takes input, process it and gives output.

Laptops, Desktop Computers are just one form of computer. Mobile phone, Calculator, Digital Watch, Digital Camera, Washing Machine etc are also computers as they also work on the given input. The difference is in their capabilities.

1.2 Does computer really work on 0 and 1?

There is no physical significance of 0 and 1 in computer system. 0 and 1 are just representation of two states of voltage levels in digital circuits. If there is a supply of high

volt in the circuit it is termed as 1 and for low volt it is 0. Computer understands only the binary language in forms of 0's and 1's. Whatever input we are given it converts to 0's and 1's, process the input and results the output in binary which is converted to human readable form.

These encoding help us to handle any sort of information through digital circuits.

1.3 What is Hardware?

Hardware is a physical device consists of machine, wires, and circuits. In Computer world, hardware is a comprehensive term for all of the physical parts of a computer, as distinguished from the data it contains or operates on, and the software that provides instructions for the hardware to accomplish tasks.

Below are the few common physical parts in a computer:

- Mother Board
- Central Processing Unit(CPU)
- Random Access Memory(RAM)
- Hard Drive

1.4 Files

A collection of data or information that has a name to identify is called a File. A computer file is a data bundle, created by a program used to refer back to saved work or support the operations of the program. There are many different types of data files:

- Text File
- Audio File
- Video File
- Program File
- Directory File

File name and extension are separated by a dot operator. File extension tells about type of the file. For example track1.mp3 is a file with extension mp3 which means it is an audio file.

Among the several file formats, we are concern about a special file format with extension exe. These files are called executable files.

Software are programs (or a collection of programs) developed by writing instructions (that need to be executed during execution) in certain programming language.

So the abstract is, you are here to write code in C language which then converted in to an exe file.

You can deliver this exe file to your client. He can enjoy your software without seeing your code. In the later segment you would understand why he can't see your programming code.

1.5 Program and Process

Program is a set of instruction to perform a particular task.

Active state of a program is called process. Running the program instructions is called process.

1.6 Software and its Types

A computer is just a machine. It can't do anything without any instructions. Set of instructions given to a computer in a precise manner to do a task is called Software. Set of instructions are written in a programming language.

Software can be classified into following two main categories or types

1. System Software:

It includes program that are necessary to operate and control a computer system.

E.g. Operating System, Device Drivers

2. Application Software: A program or set of programs designed for a specific purpose.

E.g. MS Word, Web Browser

1.7 Operating System

Operating system is essential software which provides an interface between user and machine. OS is also a program which enables the computer hardware to work with the computer software. It acts as a manager in the system, which manages hardware and software resources. It also does process management, memory management and file management.

Examples of Operating systems are windows XP, Windows 2007, Windows 2008, Linux, etc.

1.8 How a Program Executes

It is important to describe the role of few of the hardware resources during execution of program. Earlier we talked about an executable file. Here we are describing execution process of any one of such exe file.

Firstly, your program (sum.exe) file is stored somewhere in hard disk of your system. Now we clicked on it to run this program. This click event from mouse is listened by operating system and loads this file from hard disk to RAM (random access memory) using loader (a dedicated program to load content to RAM). Here it is important to note that each time when some program need to be activate, program must load in to RAM. RAM is also known as primary memory which is volatile in nature that is nothing can reside permanently. RAM contains only those programs that are active at a particular instant.

It's the job of Operating system to allocate memory for our program in RAM. First instruction goes to the processor and resides in instruction register. This instruction is decoded by control unit. Control unit sends appropriate signal to ALU. ALU performs specific task.

Registers, control unit and Arithmetic Logic Unit are part of processor. ALU is responsible to perform all kind of arithmetic, logical and shift instructions.

Again next instruction moves to processor and stored at instruction register, control unit decode it and sends a signal to ALU. Every instruction of the program executes in similar fashion in a sequential manner.

As soon as the last instruction is executed memory of the program will released from RAM.

1.9 How to develop software with C language?

Executable files are machine code which is operating system dependent and hardware dependent. To create such file we need to know about hardware specifications and OS policies. This increases extra learning overhead to the programmer. One easy solution is to learn a simple hardware independent coding language like C.

Once you learn C language you can write code (known as source code) and save it as a source file (*.c extension only). This code is not readable for machine. You need to convert it into machine code. A software known as compiler is used as a translator, which translates your source code in to object code (binary code).

Object files containing machine code are not sufficient to understand by computer, you need to link this code with predefined code needed for execution of a program stored in library files. This can be done with the help of software called linker. Linker generates an executable file. (*.exe file)

1.10 History of C language

- C language was developed by Dennis Ritchie in 1972 at AT & T's Bell Labs, New Jersey, USA.
- C language is inspired from B language which was developed by Ken Thompson.
- 80% code of UNIX operating system is written in C language.

- C is a middle level language that is a combination of high level and low level language.
- C follows procedure oriented programming approach.
- C follows top down approach.

1.11 IDE

Integrated Development Environment is a software application that provides comprehensive facilities to computer programmer for software development.

An IDE normally consist of:

- A source code editor
- Compiler or interpreter
- Build automation

- A debugger

There are' n' numbers of IDEs available for C. They are specific to operating system. These IDEs not only gives you facility to write and edit code but also provide facility to compile and link to create object file and executable file. Turbo is one of the IDE for executing C Programs.

1.12 Technical Terms

Compile: It is a process of compiling your source code and producing object code

Debug: It is an activity to remove errors in your source code. Errors are called bugs.

Error: Errors are categorized as syntax error (compile time error), Linker error, Run time error, Logical error. These are mistakes in your program which needs to be handled

Run: When you run your program it means execution of your program.

Syntax: Grammar for computer language

Turbo: Software (IDE) developed by Borland Inc.

Turbo: Software (IDE) developed by Borland Inc.

1.13 First C program

Before begin the actual journey of learning C language, let us have a look on how handsome our C program is.

```
/* Program to add two numbers*/
#include<conio.h>
#include<stdio.h>
void main()
{
    int a,b,c;
    clrscr();
    printf("Enter two numbers\n");
    scanf("%d %d", &a, &b);
    c=a+b;
    printf("Sum of %d and %d is %d", a,b,c); // printing the result
    getch();
}
```

1.14 Understanding the Program structure

- 1) The very first two lines of the program are used to include statements written in specified files. Statements starts from # symbol are known as pre-processor directives. Symbol # is followed by command **include** which tells to include content of specified files.

The files **conio.h** and **stdio.h** are known as header files, they contains declaration of predefined functions and declarations for other identifiers. The **conio.h** file contains

declaration for the functions **clrscr()** and **getch()**. Similarly, **stdio.h** contains declarations for **printf()** and **scanf()**.

We will more enlighten header files later in chapter covering functions.

- 2) C language is a block structured programming language. Pair of curly braces forms a block “{ }”.

In the above program we use a block containing all the instructions. This block is identified by a name **main()**. Execution always begin with **main()** function. The prefix of **main()** is **void**, which says it returns no value. We enlighten this thing later in functions chapter.

- 3) The line `int a, b, c;` declares variables which will be used to hold data.

clrscr() is a function used to clear the content of the output screen.

The next line is used to print a message on the screen to provide direction to the user. This is done with the help of a function **printf()**.

The function **scanf()** is used to take input from keyboard. Input data is then stored in the variables **a** and **b** which is declared previously.

Next instruction is arithmetic instruction used to calculate sum of values stored in **a** and **b**, and result is then assigned to variable **c**. The expression is same as we do calculation in mathematics.

Again function **printf()** is used to show the result to end user in an output screen.

Last line is the call to function **getch()**. The job of this function is to get a character from keyboard. This function call is required to hold the output screen, as if all the lines gets over output screen disappears immediately.

1.15 Execution of C Program

C Program executes in following four steps:

1) Creating a Program:

Developers will a create a C program with “.c” extension. The file contains source code.

2) Compiling a Program:

Program need to be compiled to get an object file which is containing the translated form of your C program. Compiler translated source code into the operating system understandable format. This is the job of compiler to compile program and converts it into object code(binary code)

3) Linking a Program:

Object file is then needed to be linked with library files needed for the execution of a program to get an executable file. The linker is used to link program with the libraries.

4) Executing a Program:

Now you can execute your software i.e. an exe file. One cannot see source code by just executing executable file.

Check Yourself:

Objective Questions:

- 1) In the context of C Programming language, what is the extension of source file?
 - a. C
 - b. CPP
 - c. EXE
 - d. OBJ
- 2) In the context of C Programming language, what is the extension of executable file?
 - a. C
 - b. CPP
 - c. EXE
 - d. OBJ
- 3) Who developed the C programming language?
 - a. Ken Thompson
 - b. Dennis Ritchie
 - c. Martin Richards
 - d. Bjarne Stroustrup
- 4) Which of the following is an operating system?
 - a. RAM
 - b. ALU
 - c. Windows 7
 - d. MS Office
- 5) Which of the following is not a job of an operating system
 - a. Process Management
 - b. Memory Management
 - c. Compilation of C program
 - d. File Management
- 6) What is the full form of IDE?
 - a. Internet Data Editor
 - b. Integrated Development Environment
 - c. Internal Data Editor
 - d. Integrated Development Engine
- 7) When C language was developed?
 - a. 1966
 - b. 1972
 - c. 1979
 - d. 1983
- 8) C is a..... (more than one answer correct)
 - a. Middle Level Language
 - b. Procedure Oriented Language
 - c. Block Structured Programming Language
 - d. Object Oriented Programming Language
- 9) Which of the following is not a part of processor
 - a. ALU
 - b. RAM
 - c. Registers
 - d. Control Unit
- 10) What is the job of linker?
 - a. Compile the source code
 - b. Execute program
 - c. Link header files with source code
 - d. Link object codes and library code

Explore and Answer the following

- 1) What is the difference between Compiler and Interpreter
- 2) What is file?
- 3) Why C is called middle level language?
- 4) Name any five well known software written in C Language
- 5) Explain IDE.

Chapter 2: Start Learning C

Content

- 2.1 Start Learning
- 2.2 Constituents of C
- 2.3 Identifiers
- 2.4 Constant
- 2.5 Variable
- 2.6 Keywords
- 2.7 Instructions and its types
- 2.8 Data type declaration instruction
- 2.9 Data type Modifiers
- 2.10 Input / Output Instruction
- 2.11 Escape Sequence
- 2.12 Input Instruction

2.1 Start Learning

As a matter of fact any language whether it is natural language (like Hindi, English, French, etc) or computer programming language, one has to learn basic constituents first.

For example alphabets are basic constituents of English language then learn to combine these alphabets into words which in turn to sentences and then paragraph.

In a similar fashion we have to learn basic constituents of C language. The basic constituents are alphabets, digits, special symbol which will combine called basic letters.

The order of our journey contains:

Basic Letters - > Identifiers - >Instructions - >Program

2.2 Constituents of C

Language writing includes letters as:

Alphabets	A to Z a to z
Digits	0 to 9
Special Symbol	~ ! @ # % ^ & * () _ - + = \ { } [] : ; " ' <> , . ? /

2.3 Identifiers

Combination of basic letters forms Identifiers. Identifiers are smallest unit in the program that has some meaning. We know that compiler will be reading our source code to translate it in object code. So compiler should know about all the words of our program. Few words are predefined to compiler and rest need to be declared by the programmer. These words are identifiers as they can be identified by compiler.

Identifier includes **constants, variables, keywords, macros, functions** etc.

Here we are going to understand three of these; they are constants, variables and keywords.

2.4 Constant

Software is developed to handle information. This information is called constant. Sometimes it is termed as data. Value of constant does not change.

Constants can be categorized as Primary and secondary constants

2.4.1 Primary Constant:

Primary constants are also known as fundamental constants or basic constants. They are of three types:

1. Integer constant
2. Real constant
3. Character constant

1. Integer Constant:

All numbers either positive or negative without decimal point are Integer Constant
For example: 35, -20, 0, 123 are valid Integer constant.

2. Real Constant:

All numbers either positive or negative with decimal point involved in it are real constant.

For example: 3.14, -43.56, 3.0 are valid Real constant.

3. Character Constant:

All character symbols are character constants if they are enclosed in single quotes and of unit length.

For example: 'A', 'b', '+', '3', ''

2.4.2 Secondary Constants:

Secondary constants are also known as derived constants, as they are derived from primary constants.

Arrays, strings, pointers, structures, union, enumerators are secondary constants.

We will see them in great detail in later chapters.

2.5 Variables

Variables are the name of memory locations where we store our data. When C program is under execution, operating system allocates memory to store instructions of the program and also allocates space to store data.

Programmer has to decide how much space he is required with to store data. He also specifies the names of memory locations in his program code. These names are called variables. These locations can contain integer, real and character constants.

We can change its value according to programming requirement.

Rules for constructing variable name:

1. Variable name may contain alphabets, digits or underscore.
2. No other symbol or blank is allowed.
3. A valid variable name never starts with digit.

2.6 Keywords

Keywords are predefined words. They are also known as reserved words. Their meaning is known to the compiler.

ANSI (American National Standards Institute) announces 32 keywords in C language, but it may vary with compilers.

Keywords are predefined and we will see their meaning in subsequent chapters.

Here is the list of keywords:

Auto	break	case	char	continue	const	int
Do	default	enum	extern	for	float	register
double	else	goto	long	near	Return	struct
Huge	far	if	sizeof	union	unsigned	while
static	short	signed				

These keywords cannot be used as variable or function names as they already have some meaning known to compiler.

2.7 Instructions and its types

A program is a set of instructions. Instructions are also known as commands or statements. They are the sentences in program. Just like any other language, C also provides variety of instructions.

2.7.1 Types of Instructions:

Data type declaration instruction

Input/Output instruction

Arithmetic instruction

Control instruction

2.8 Data type declaration instruction

From the list of keywords few are designed to use in declaration instruction. They are required to declare variables and functions. These keywords are

int, float, char, double, void

Here we are only concern with variable declaration. Declaration of functions will be deal in later chapter.

Consider the following statements

```
int x=5, y;  
float k=2.34;  
char a='y', ch;
```

These are examples of data type declaration instruction. Compiler when read these instructions, it would get aware of four things:

1) Name of the variables

Here names are x ,y, k, a, ch

2) Size of memory block

Size of int type block is 2 bytes, size of float block is 4 bytes, size of char block is 1 byte, size of double block is 8 bytes

3) Type of content

int variable can contain integer constant, float and double variable can contain real constant and char variable can contain character constant.

4) Value in variable

x contains 5, k contains 2.34 and a contains 'y'.

Variables y and ch are not initialized but they are not empty. They contain unpredictable values which are of no use in the program, so we call them garbage values.

2.9 Data type Modifiers

Modifiers are the keywords used to modify the properties of data type.

Modifiers are **short, long, signed, unsigned**.

Here is the list of all possible combination of use of modifiers with data type:

Create variables to store integers:

int, short int, signed int, long int, unsigned int, unsigned short int, unsigned long int.

Create variables to store real constant

float, double, long double

Create variables to store character constant

char, unsigned char

Modifiers affect size of data type (short and long), data range (signed and unsigned)

Type	Size	Format specifier	Content Type	Range
unsigned char	8 bits	%c	Character	0 to 255
Char	8 bits	%c	Character	-128 to 127
unsigned int	16 bits	%u	Integer	0 to 65,535
short int	16 bits	%d	Integer	-32,768 to 32,767
Int	16 bits	%d	Integer	-32,768 to 32,767
unsigned long	32 bits	%lu	Integer	0 to 4,294,967,295
Long	32 bits	%ld	Integer	-2,147,483,648 to 2,147,483,647
Float	32 bits	%f	Real	1.1754*(10^-38) to 3.4028* (10^+38)
Double	64 bits	%lf	Real	2.2250 * (10^-308) to 1.7976 * (10^+308)
long double	80 bits	%Lf	Real	3.4 * (10^-4932) to 1.1 * (10^4932)

2.10 Input / Output Instruction

2.10.1 Output Instruction

User interaction with a program on machine required input/output device. Here, we preferably use keyboard as input device and monitor as output device.

printf()

printf() is a pre-defined function used to display messages on monitor. Wherever function name is used it must be suffixed with ()

Syntax:

printf("format string", variable list);

Format string is a message that you want to print on the screen.

Variable list is optional. We can also write expression in place of variable list.

Example:

```
printf("Saurabh Shukla");
printf("Hello SCA");
```

Write your first program

```
main()
{
    printf("Hello SCA");
}
```

Step 1: Save this program as first.c

Step 2: Compile it

Step 3: Run

Your program executes in a flash and you would not able to see its output.

Press ALT + F5 to see output window or modify your program as:

Modified Program:

```
main()
{
    printf("Hello SCA");
    getch();
}
```

When your program runs, first instruction prints a message Hello SCA.

Second instruction is a call to a function **getch()** whose job is to take one character from keyboard. Until you press any key this line wouldn't finish its job. As call to **getch()** is the

last line of the program, termination of program depends on when you enter a character. This let you see output window till you press any key.

You probably wonder why your older output is still on the screen every time you run your program. This is due to your consecutive execution of program using same output window. This problem can be solved by using clrscr() before printf().

Call to function clrscr() erase content of output screen.

Modified program:

```
main()
{
    clrscr();
    printf("Hello SCA");

    getch();
}
```

Write a program to display values of two variables on the monitor. Declare these two variables of any type and assigned some values to them.

```
main()
{
    int a=3,b=56;
    clrscr();
    printf("%d %d",a,b);
    getch();
}
```

Output:

3 56

In the above example **printf** is use to display contents of variables a and b. Notice that %d is a special symbol called format specifier. They are used to specify format of data to be printed. In the format string %d will be replaced by values of the variables.

We can write expressions in place of variable list.

```
main()
{
    int a=3,b=56;
    clrscr();
    printf("%d %d %d",a,b, a+b);
    getch();
}
```

Output: 3 56 59

Notice that printf contains three format specifiers (%d) as there are three expressions in the variable list. First two are simply variables and third one is an expression which first solved then the result gets printed in the place of third %d.

2.11 Escape Sequences

Escape sequences are special symbols prefixed with a back slash (\). These symbols are used in printf to provide a formatted output.

Here is the list of escape sequences:

Escape Sequence	Name	Meaning
\a	Alert	Produces an audible or visible alert.
\b	Backspace	Moves the cursor back one position (non-destructive).
\f	Form Feed	Moves the cursor to the first position of the next page.
\n	New Line	Moves the cursor to the first position of the next line.
\r	Carriage	Return Moves the cursor to the first position of the current line.
\t	Horizontal Tab	Moves the cursor to the next horizontal tabular position.
\v	Vertical Tab	Moves the cursor to the next vertical tabular position.
\'		Produces a single quote.
\”		Produces a double quote.
\\\		Produces a single backslash.
\0		Produces a null character.
\ddd		Defines one character by the octal digits (base-8 number).
Multiple characters		may be defined in the same escape sequence, but the value is implementation-specific.
\xdd		Defines one character by the hexadecimal digit (base-16 number).

Consider the following program

```
main()
{
    int a=3,b=56;
    clrscr();
    printf("%d\n%d",a,b);
    getch();
}
```

Output

3
56

Here output comes in two lines. Notice \n in the printf , first value (3) gets printed then \n moves cursor to the next line and then 56 gets printed.

Use list of different escape sequences in your program and observe the effect.

2.12 Input Instruction

Default input device is keyboard. Input instruction makes possible to input data from keyboard.

scanf()

scanf() is a predefined function. It is used to input data from keyboard and stores in variables of the program.

Syntax:

```
scanf("format specifier", address of variable);
```

Example:

```
main()
{
    int a;
    scanf ("%d", &a);
}
```

In this program, we have declared a variable a of type int. **scanf** makes possible one integer value through keyboard. scanf receives this value and stores it in variable a.

Here it is important to notice address of operator (&) before variable a. don't forget to put this operator before variable name. We will understand this operator in great detail in later chapter.

Multiple input:

```
main()
{
    int a;
    float y;
    scanf ("%d%f", &a, &y);
}
```

For float variable format specifier should be %f. Here we can input two values, first an integer and second one is float.

Write a program to add two numbers.

```
main()
{
    int a,b,c;
    clrscr();
    printf("Enter two numbers");
    scanf ("%d%d", &a, &b);
    c=a+b;
    printf("Sum is %d", c);
    getch();
}
```

Output

Enter two numbers5

6

Sum is 11

Now you can use **scanf** for user input. We will again study **scanf** in great detail during functions in later chapter.

Check Yourself

Objective Questions:

- 1) Which of the following is an integer constant?
 - a. -3.4
 - b. 'a'
 - c. '5'
 - d. 34
 - 2) Which of the following is a character constant?
 - a. -44
 - b. 2.1
 - c. '3'
 - d. "a"
 - 3) Which of the following is a secondary constant?
 - a. Integer
 - b. String
 - c. Character
 - d. Real
 - 4) Which is a valid variable name?
 - a. A-1
 - b. 2d
 - c. S.i
 - d. B_2
 - 5) Which of the following is not a keyword in C language?
 - a. static
 - b. double
 - c. this
 - d. void
 - 6) Which of the following is a keyword in C language?
 - a. class
 - b. new
 - c. extern
 - d. public
 - 7) What is the full form of ANSI?
 - a. American National Standards Institute
 - b. American National Statistical Institute
 - c. American Numbering Standards Institute
 - d. American Native Standard Institute
 - 8) Which of the following is a real constant?
 - a. "3.4"
 - b. -3.0
 - c. '4'
 - d. None of the Above
 - 9) Which of the following is correct statement?
 - a. A valid variable name can start with a digit

- b. Keywords are words that should be declared by programmer
 - c. Identifier is the smallest unit of the program
 - d. C is a case insensitive language
- 0) Which of the following keyword is not a data type?
- | | |
|-----------|-----------|
| a. void | b. static |
| c. double | d. char |
- 11) float variable consumes _____ bits in memory
- | | |
|-------|------|
| a. 16 | b. 4 |
| c. 32 | d. 8 |
- 12) Which of the following is incorrect statement?
- a. A variable name can be a combination of alphabet, digit and underscore
 - b. Keyword char can also be written in upper case (CHAR), without losing its meaning
 - c. Every statement in C language is terminated by semicolon
 - d. A program is a set of instruction

Write following programs:

- 1) Write a program to print your name on the first line and surname on the second line of console window.
- 2) Write a program to print your name on the screen, as soon as a key is pressed name should disappear from the screen, again on pressing any key, surname gets printed.
- 3) Write a program to input a character from keyboard and display its ASCII code.
- 4) Write a program calculate simple interest. Data should be supplied by user through keyboard.
- 5) Write a program to calculate area of circle.
- 6) Write a program to input three data, each of type int, char and float. Also display them.
- 7) Write a program to calculate average of three numbers. Numbers are taken from keyboard.
- 8) Write a program to calculate volume of a cuboid. Data should be taken from keyboard

Explore and Answer the following:

- 1) Write a program to print \n on the screen.
- 2) Write a program to print %d on the screen
- 3) Write a program to print “SCA”
- 4) Differentiate between getch() and getche()

- 5) Write a program to input a character from keyboard and display that character. Use getch() in place of scanf().
- 6) Write a program to print your name at the centre of the screen
- 7) Write a program to check whether the action statement can be written before declaration statement.

Chapter 3: Arithmetic Instruction

Content

- 3.1 Arithmetic Instruction
- 3.2 Unary Operators
- 3.3 Arithmetic Operators
- 3.4 Bitwise Operators
- 3.5 Relational Operators
- 3.6 Logical Operators
- 3.7 Conditional Operators
- 3.8 Assignment Operators

3.1 Arithmetic Instruction

Arithmetic instruction is used to manipulate data using operators

Here is the list of operators

Operator	Description	Associativity
()	Parentheses (grouping)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Prefix, Postfix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(<i>type</i>)	Cast (change <i>type</i>)	
*	Dereference	
&	Address	
sizeof	Determine size in bytes	

* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= =	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

To understand operators in a better way we put them in 8 groups. This also helps you in learning precedence rule of operators. Each group contains operators of same nature. So it would be better if you learn the group names in the same order as listed below.

- Unary Operators
- Arithmetic Operators
- Bitwise Operators
- Relational Operators
- Logical Operators
- Conditional Operators
- Assignment Operators
- Miscellaneous Operators

3.2 Unary Operators (+, -, ++, —, sizeof())

Operators required operands to perform its operation. Unary operators are those who required with only single operand to perform its task.

Unary + and -

These operators are not addition and subtraction, rather they are unary + and – that shows direction of data. For example -3, +4, -345 etc

Increment Operator ++

Increment operator (++) increase value of variable by one. Let us assume we have an int variable x containing 4.

```
int x=4; //declaration of variable x and assign 4 to it.  
x++; //Value of variable x is incremented by one and become 5  
++x; // Value of variable x is incremented by one and become 6
```

Here notice that we have two conventions to use increment operator. You can put operator before operand or after operand. If it is after operand known as post increment, otherwise known as pre increment. Both these convention have same effect over operand x as they increase its value by 1. But these two conventions differ in priority rule. Pre increment has the highest priority among all the operators. Post increment has the lowest priority among all the operators.

sizeof() operator

sizeof() operator is used to know size of data type, variable or constant.

Example :

```
main()  
{  
    int x,y;  
    float k;  
    char ch;  
    double d;  
    x=sizeof(float); // sizeof returns 4 and assigned to x  
    x=sizeof(char); // sizeof returns 1 and assigned to x  
    x=sizeof(int); // sizeof returns 2 and assigned to x  
    x=sizeof(double); // sizeof returns 8 and assigned to x  
    x=sizeof(d); // sizeof returns 8 and assigned to x  
    x=sizeof(k); // sizeof returns 4 and assigned to x  
    x=sizeof(ch); // sizeof returns 1 and assigned to x  
    x=sizeof(y); // sizeof returns 2 and assigned to x  
    x=sizeof(45); // sizeof returns 2 and assigned to x  
  
    x=sizeof(23.67); // sizeof returns 8 and assigned to x  
    x=sizeof('a'); // sizeof returns 2 and assigned to x  
}
```

Always remember that the size of real constant is 8 bytes and character constant is 2 bytes. Character constants are internally treated as integers. Each character constant has a unique integer code called **ASCII** code. **ASCII** stands for American **Standard Code for Information Interchange**.

Associativity rule for unary operators is right to left. Which means when more than two unary operators of same priority appears in a single arithmetic expression, they will be solved from right to left.

Regular ASCII Chart (0 to 127)

Extended ASCII Chart (128 to 255)

3.3 Arithmetic Operators (+,-,*,/,%)

+,- and * operators are same as they are in mathematics. + is used to add two numbers, - is used to subtract two numbers and * is used to multiply two numbers.

Divide operator (/)

```
main()
{
    int x;
    x=5/2;
    printf("%d",x);
}
```

Output:

2

Compiler understands numbers in two categories one is integer and second is real. We have used both the operands integer in above program. When two integers operate, result is always an integer. So 5/2 yield 2 and not 2.5

To get result 2.5 we need to make at least one of the operand real.

```
main()
{
    int x;
    x=5/2.0;
    printf("%d",x);
}
```

Output:

2

Here we made one operand real but still result is 2. It happens because variable x is of type int and could not hold real constant. So choose float variable.

```
main()
{
    float x;
    x=5/2.0;
    printf("%d",x);
}
```

Output:

2

Oh man! Again the output is 2. Although we made one of the operand real and variable is also of type float, but notice the format specifier used in printf to display content of x, which is wrong. You have to use %f for float variable.

```
main()
{
    float x;
    x=5/2.0;
    printf("%f",x);
```

}

Output:

2.500000

Modulo operator (%)

Modulo operator gives remainder in result. It cannot work for real constants.

```
main()
{
    int x;
    x=5%2;
    printf("%d", x);
}
```

Output is:

1

When 5 is divided by 2 remainder comes 1.

Note: Modulo operator is good to check divisibility of a number. When remainder is 0, number is divisible.

Precedence of *, / and % is same but higher than + and -.

Precedence of + and – is same.

Associativity rule for arithmetic operators is left to right.

3.4 Bitwise operator (&, |, ^, ~, >>, <<)

Bitwise operators work on binary numbers that is bits. You should know how to convert a number from decimal to binary and from binary to decimal.

Nature of operators

Bitwise AND (&)

0&0 is 0

0&1 is 0

1&0 is 0

1&1 is 1

Bitwise OR (|)

0|0 is 0

0|1 is 1

1|0 is 1

1|1 is 1

Bitwise XOR (^)

0^0 is 0

0^1 is 1

1^0 is 1

1^1 is 0

Bitwise NOT (~)

~ 0 is 1

~ 1 is 0

```
main()
{
    int x;
    x=5&12;
    printf("%d",x);
}
```

Output is:

4

Bitwise AND applies on 5 and 12. We need to convert them in binary.

5 = 00000000 00000101

12 = 00000000 00001100

& _____
4 = 00000000 00000100

Shift Operators

Right shift >>

Left shift <<

```
main()
{
    int x;
    x=12>>2;
    printf("%d",x);
}
```

Output:

3

Convert 12 into binary and shift bits to their right 2 times. This makes last two bits out and two new bits (always 0) append at the left.

12 = 00000000 00001100

Right shift two times

3 = 00000000 00000011

```
main()
{
    int x;
    x=12<<2;
    printf("%d",x);
}
```

Output :

48

Convert 12 into binary and shift bits to their left 2 times. This makes last two left most bits out and two new bits (always 0) append at the right.

```
12=00000000 00001100  
Left shift two times  
48=00000000 00110000
```

3.5 Relational Operators (`<`, `>`, `<=`, `>=`, `==`, `!=`)

Relational operators are always evaluated as true or false. True is 1 and false is 0.

```
main ()  
{  
    int x;  
    x=5<3;  
    printf ("%d", x);  
}
```

Output is:

0

As the relation $5 < 3$ is false, x will contain 0 as a result of relational operator.

Priority of `<`, `>`, `<=`, `>=` is higher than

`==` and `!=`

Associativity is left to right.

3.6 Logical Operators (`&&`, `||`, `!`)

Logical operators (`&&` and `||`) are used to combined two expressions.

They work as:

C1 && C2	Result
T && T	T
T && F	F
F && X	F

C1 C2	Result
F F	F
F T	T
T X	T

Example:

```
main ()  
{  
    int x;  
    x= 5>3&&4<0;  
    printf ("%d", x);  
}
```

Output is:

0

Here, two conditions $5 > 3$ and $4 < 0$ are combined to form a single condition using `&&` operator as $5 > 3 \&\& 4 < 0$. Condition one is evaluated as TRUE as 5 is greater than 3. Since condition one is TRUE condition two is tested and evaluated as FALSE as 4 is not less than 0. According to the above chart, T&&F is treated as FALSE thus 0 is stored in x.

Logical NOT (`!`)

`!` operator is used to invert the truth value.

```
main ()
```

```

{
    int x;
    x=! (5>4);
    printf("%d", x);
}

```

Output is:

0

$5>4$ is evaluated as TRUE, NOT (!) operator invert this result and make it FALSE.

Thus 0 is stored in x.

3.7 Conditional Operator (?:)

Conditional operator is also known as ternary operator. We will discuss it in our next chapter control instructions.

3.8 Assignment operators (=, +=, -=, *=, /=, %=)

Assignment operators are used to assign values to variable

```

main()
{
    int x;
    x=4;
}

```

In the above code we used assignment operator to store 4 in x.

```

main()
{
    int x=5;
    x+=4; // same as x=x+4
}

```

Here, $+=$ is also an assignment operator, but used to store data after small manipulation. First 4 is added to the content of x and then assigned to x.

The same operation can be performed using = and + operator as $x=x+4$, but this expression contains two operators which requires precedence rule during evaluation.

Although, generated results are same that is x is containing 9.

```

main()
{
    int x=3, y=3;
    x*=4+3;
    y=y*4+3;
}

```

Now you can understand difference between the two expressions. First $x*=4+3$, $=$ is operated first as it has higher priority than $*$. So $4+3$ becomes 7. x is then multiply to 7 to give result 21. This 21 is getting stored in x.

Second expression $y=y*4+3$ which is thought as the substitute of first expression but it does not work in similar fashion. $*$ has the highest priority then $+$ and then $=$ comes at the last. Content of y is multiplied by 4 which give 12. 12 is added to 3 which becomes 15 and get stored in y.

Other operators $-=$, $/=$ and $\%=$ are works in a similar fashion.

Check Yourself

Objective Questions

```

main()
{
    int x=3,y;
    y=3 + sizeof 4*x+5;
    printf("x=%d y=%d",x,y);
}

```

- a. x=3 y=5 b. x=3 y=14
 c. x=3 y=10 d. x=3 y=2
- 8) int a=10, b ;
 b=a++ + ++a;
 printf ("%d,%d,%d,%d",b,a++,a,++a);
 What will be the output of the above code when executed?
 a. 12,10,11,13 b. 22,10,11,13
 c. 22,11,11,11 d. 22,13,13,13
- 9) What is the output of the following program?
- ```

void main()
{
 int c=08;
 printf("%d",c);
}

```
- a. 8                              b. 0  
  c. Compile time error              d. 10
- 10) The statement printf("%c", 100); will print?  
  a. prints 100                      b. print garbage  
   c. prints ASCII equivalent of 100 d. None of the above

### **Write following programs**

- 1) Write a program to swap two numbers.
- 2) Write a program to display last digit of a number. Number is entered through keyboard.
- 3) Write a program to calculate sum of the digits of three digit number.
- 4) Write a program to calculate square of a number
- 5) Write a program to print profit and profit percentage. Selling price and cost price is given by user.

### **Explore and Answer the following**

- 1) Write a program to swap two numbers without using third variable.
- 2) Write a program to swap two numbers without using third variable and arithmetic operators.
- 3) Can we calculate the sizeof void?
- 4) What should be the result of  $3.5\%2$  ?
- 5) What should be the result stored in variable x after execution of the statement  $x=\sim 21$ 
  - a.  $x*=3+4;$
  - b.  $x=x*3+4$

# Chapter 4: Control Instruction

## Content:

- 4.1 Control Instruction
- 4.2 Decision Control Instruction
- 4.3 If
- 4.4 if-else
- 4.5 Conditional operator
- 4.6 Nested if else
- 4.7 else if ladder

## **4.1 Control Instruction**

Program is a set of instructions. We know that each instruction of the program is executed by processor. Processor executes instructions one by one. Whatever we write in our program will be executed in the same order as they appear in the program. So we can say processor executes instructions in a sequential manner. At a particular instant processor is executing some line of code, we say control of processor is on that line. Processor's control moves from one line to another. We say this movement of processor control goes in sequence

### **Types of control instruction**

- Decision control instruction
- Iterative control instruction
- Switch case control instruction
- goto instruction

## **4.2 Decision control instruction**

Decision control instruction is also known as selection control instruction. As the name implies the job of selection control instruction is to select a set of code for execution on the basis of some condition.

We can implement decision control instruction in three ways:

- if
- if-else
- conditional operator (Ternary Operator)

### **4.3 if**

#### **Syntax:**

```

main()
{

 if(some condition)
 {
 Statement1;
 Statement2;

 }
 ...
}

```

**if** is a keyword which let compiler to identify decision control instruction. Immediately after if some condition is there. This condition is any valid expression in C. If the result of expression is non-zero it is considered as TRUE otherwise FALSE.

Immediately after this condition there is a block of code. Since this block is immediately after if, it is known as **if block**. Whatever we write in if block will be execute only when condition is TRUE.

When condition is false control skip **if block** and execute statements written after **if block**.

**Example:**

```

main()
{
 int marks;
 printf("Enter marks ");
 scanf("%d", &marks);

 if(marks>=33)
 {
 printf("You are PASS");
 }
 if (marks<33)
 {
 printf("You are FAIL");
 }
}

```

Sample Output:

```

Enter marks 45
You are PASS

```

Sample Output:

```

Enter marks 23
You are FAIL

```

In this program output depends on the value given by user. Variable marks hold the value entered by user. We have used two **if** statements. In the first **if** statement we use the condition  $\text{marks} \geq 33$ , thus if the marks are greater than or equal to 33 condition becomes TRUE, so if block executed, otherwise if block is skipped.

Whatever may the result of first **if** condition, control has to reach second **if** statement.

If marks are less than 33 condition will be TRUE and execute if block otherwise if block is skipped.

#### 4.4 if-else

## Syntax

```
main()
{

 if(some condition)
 {
 Statement1;
 Statement2;

 }
 else
 {
 Statement1;
 Statement2;

 }
 ...
}
```

This is similar to **if** but the **else** block is new add on. If the condition of **if** is TRUE **if block** will be executed and if the condition of **if** is FALSE **else block** will be executed.

It is important to mention that only one from the two blocks (if block and else block) can be executed as the condition may have only two results, TRUE or FALSE

You can use if statement without else block but else must have paired with if.  
Else block should appear immediately after if block otherwise an error occurred during compilation.

Example:

```
main()
{
 int marks;
 printf("Enter marks ");
 scanf("%d", &marks);
 if(marks>=33)
 {
 printf("You are PASS");
 }
 else
 {
 printf("You are FAIL");
 }
}
```

The same program was discussed using only if statements. This one is refined version of program and hence better than that of the previous one.

Notice that, there is only one condition need to be evaluated, if the condition is TRUE if block will work otherwise else block will be executed.

Note: If there is only one statement in if block then mentioning block using curly braces is optional. Same rule is applied to else block.

## 4.5 Conditional operator ( ?: )

It is also known as ternary operator which means operator need three operands to perform its operation.

#### Syntax:

```
Expression1 ? expression2 : expression3;
```

Expression 1 is a condition, which is first evaluated as TRUE or FALSE. If the condition is TRUE executes expression2 otherwise execute expression3.

Conditional operator works similar to if-else, but we do not have to use keyword if and else.

Example:

```
main()
{
 int x,y;
 printf("Enter two numbers");
 scanf("%d%d",&x,&y);
 x>y ? printf("%d is greater",x) : printf("%d is greater",y);
}
```

Output:

```
Enter two numbers45
33
45 is greater
```

In this program user enters two numbers which is then get stored in x and y. Notice the last line of the program that is conditional operator, which is used to select one from the two printf() statements. If  $x > y$  then value of x get printed otherwise value of y is printed.

```
main()
{
 int x,y,max;
 printf("Enter two numbers");
 scanf("%d%d",&x,&y);
 max=x>y ? x : y;
 printf("Greater number is %d",max);
}
```

Output is:

```
Enter two numbers85
100
Greater number is 100
```

In this program conditional operator is used to select one from x and y to assign value of either x or y in variable max.

#### 4.6 Nested if-else

When if-else statements written in **if block** or **else block**, it is called nested if-else.

Example:

Program to find greater among three numbers:

```
main()
{
 int a,b,c;
 printf("Enter three numbers: ");
```

```

scanf("%d%d%d", &a, &b, &c);
if(a>b)
{
 if(a>c)
 printf("%d is greater",a);
 else
 printf("%d is greater",c);
}
else
{
 if(b>c)
 printf("%d is greater",b);
 else
 printf("%d is greater",c);
}

```

In the above program if a is greater than b then if-block is executed otherwise else block is executed. Notice that another if-else reside inside if-block and else-block. This sort of structure is known as nested if-else.

## 4.7 else if ladder

### Syntax:

```

if()
{

}

else if()
{

}

else
{

}

```

### Example:

```

main()
{
 int year;
 printf("Enter a year");
 scanf("%d", &year);
 if(year%4!=0)
 printf("Not a Leap year");
}

```

```

else if(year%100!=0)
 printf("Leap year");
else if(year%400!=0)
 printf("Not a Leap Year");
else
 printf("Leap Year");
}

```

The above program is to check whether a year is leap year or not. You can analyze how we used if-else ladder in the above program

## **Check Yourself:**

### **Objective Questions:**

- 1) What is the output of the following code?

```

main()
{
 if(3>4)
 printf("3");
 printf("4");
 else
 printf("7");
}

```

- a. 7
- b. 34
- c. 347
- d. Error

- 2) What is the output of the following program?

```

main()
{
 int x=0, y= -1;
 if(x++&&y++)
 printf("%d %d",x,y);
 else
 printf("%d %d",y,x);
}

```

- a. 0 -1
- b. 1 -1
- c. 1 0
- d. -1 1

- 3) Find the output of the following program

```

main()
{
 int x=0, y= -1;
 if(++x&&++y)
 printf("%d %d",x,y);
 else
 printf("%d %d",y,x);
}

```

- a. 1 0
- b. 0 1
- c. 1 -1
- d. -1 1

- 4) Find the output of the following program

```
main()
{
 float a=0.7;
 if(a<0.7)
 printf("Dennis");
 else
 printf("Ritchie");
}
```

- a. Error                          b. Dennis  
c. Ritchie                        d. Dennis Ritchie

- 5) Find the output of the following program

```
main()
{
 int x=5;
 if(x=4)
 printf("%d", x+1);
 else
 printf("%d", x-1);
}
```

- a. 6                              b. 4  
c. 5                              d. 3

- 6) Find the output of the following program

```
main()
{
 int x=3,y=5;
 if(x++ || ++y)
 printf("%d %d", ++x, x++);
 printf(" %d %d", ++y, y=++x);
}
```

- a. 6 4 8 7                      b. 5 5 6 7  
c. 5 6 7 7                      d. 5 6 7 8

- 7) Find the output of the following program

```
main()
{
 int x = 0, y = -1;
 if(x++ || y++ && ++x)
 printf("%d %d", x, y);
 else
 printf("%d %d", y, x);
}
```

- a. 2 0                            b. 0 2  
c. 1 -1                         d. -1 1

- 8) Find the output of the following program

```

main()
{
 int x = 1,y = 2,z=0;
 if(x!=y>z) printf("1");
 else printf("0");
}

```

a. Error                      b. 0  
c. 1                            d. 10

9) Find the output of the following program

```

main()
{
 int a=210,b=120;
 if(a>b) printf("%d",a=a-b);
 if(b>a) printf("%d",b=b-a);
 if(a>b) printf("%d",a=a-b);
 if(b>a) printf("%d",b=b-a);
}

```

a. 903060-30                b. 903060  
c. 906030-30                d. 906030

10) Find the output of the following program

```

main()
{
 int a=210,b=120,c;
 c=a>b?1,2,3:2,5,6,7;
 printf("%d",c);
}

```

a. 1                            b. 3  
c. 2                            d. Error

### **Write following programs:**

- 1) Write a program to find greater between two numbers.
- 2) Write a program to check whether a given number is divisible by 5 or not.
- 3) Write a program to check whether a given number is even or odd.
- 4) Write a program to check whether a given number is positive or non positive.
- 5) Write a program to check whether a given number is positive negative or zero.
- 6) Write a program to find the greatest among 3 numbers.
- 7) Write a program to find greatest among 3 numbers (use ternary operator)
- 8) Write a program to check whether a given character is digit, uppercase alphabet, lower case alphabet or any other character
- 9) Write a program that takes marks of five subjects from user. Display result as PASS or FAIL. If candidate is passed then print its percentage and division.
- 10) Write a program check whether a given year is a leap year or non leap year.

### **Explore and Answer the following:**

- 1) Write a program to print your name without using semicolon ( ; ).
- 2) Write a program to check whether a given number is even or odd without using modulo (%) operator.

# Chapter 5: Iterative Control Instruction

## Content

- 5.1 Iterative Control Instruction
- 5.2 While
- 5.3 Do-while
- 5.4 For
- 5.5 Nested loop
- 5.6 The keyword break
- 5.7 The keyword continue
- 5.8 Switch case control Instruction
- 5.9 Menu driven Programming
- 5.10        Goto control Instruction

### **5.1 Iterative Control Instruction**

Iterative control instruction is also known as repetitive control instruction or loop.  
Sometimes it is desirable to execute same statement again and again. This can be done with the help of loops

There are three ways to implement loops in C language:

- while
- do-while
- for

### **5.2 while loop:**

#### Syntax of while

```
main()
{

 while(condition)
```

```

{
 ...
 ...
}
...
}
```

Syntax of while is similar to if. In the case of if, when the condition is TRUE control moves inside if block and execute statements of if-block. After executing if-block control moves to the statement written immediately after if-block (outside if-block).

In the case of **while**, when the condition is TRUE control moves inside **while-block** and execute statements of **while-block**. After executing **while-block** control does not moves to the statement written immediately after while-block rather it goes back to the condition of while block. This condition will be checked again and if it is again TRUE control moves again inside while-block. This repeats till the condition becomes FALSE. **while** loop executes the set of statements until the condition is true or non zero value. Condition can be any valid expression also.

**Example:**

```

main()
{
 int i=1;
 ...
 while(i<=5)
 {
 printf("SCA ");
 i++;
 }
 printf("Out of loop");
}
```

**Output:**

SCA SCA SCA SCA SCA Out of loop

In this example, printf inside while loop is executed 5 times. You have to think about three things to control loop execution. They are initialization, termination condition and flow. Here, i is used to control loop. In the very first line of the main function i is initialized by 1. So when the condition is evaluated first time it is interpreted as  $i \leq 5$ , thus it is TRUE. Control enters in while block and execute printf statement. After printf i is incremented by 1. This is important as we want to execute loop body only five times. Initially i was 1 and in each iteration it is incremented by 1, this makes i to reach a value where condition  $i \leq 5$  becomes FALSE. If no termination condition is defined under while loop you get trapped in an infinite loop.

### 5.3 do-while loop:

#### Syntax of do-while loop

```

main()
{
 ...
 ...
```

```

do
{

} while(condition);

...
}

```

Learn a new keyword **do**. The code block that has to put in loop is prefixed with keyword **do** and post-fixed with keyword **while** along with termination condition. Remember to put semicolon after while.

**do-while** works similar to **while** but the only difference is earlier is executed at least once. Since in while loop condition is evaluated first then goes into loop body, on the other hand in **do while** loop first control enters in loop body then condition will be checked. This makes possible to control enters in loop body even though the condition is false. Although once the condition is checked FALSE loop is terminated.

**Example:**

```

main()
{
 int i=5;
 do
 {
 printf("SCA");
 }while(i>6);

getch()
}

```

**Output:**

SCA

When we execute the above program SCA is printed in output screen once as the printf statement will get executed one time and then the condition will be check which will return FALSE as 5 is not greater than 6. If the above program is written by using while loop at the very first time condition will return FALSE and SCA will not get printed in output screen. do while loop is used in case where we want to execute the condition at least one time.

#### **5.4 for loop:**

##### Syntax of for

```

main()
{

 for(; ;)
{
 ...
}

```

```
 }
 ...
}
```

for loop is programmers choice as it contains initialization, termination and flow at same place. Notice the two semicolons inside for's round braces, these are part of syntax, hence should always be mentioned.

Two semicolons create three sections. First section is used for initialization, second section is used for termination condition and third section is used to mention flow.

```
for(initialization, condition, increment)
{

}
```

First the variable gets initialized, and then the condition is getting checked. If the condition returns TRUE the statements (body of the loop) get executed and then increment the value of the variable. Again the control transfer to check the condition. If the condition returns FALSE no statement will get executed inside the for body and transfer the control to the first statement outside the loop.

Example:

```
main()
{
 int i;
 for(i=1;i<=5;i++)
 {
 printf("\n Number = %d", i); // Statement 1
 }
 getch();
}
```

Output:

```
Number = 1
Number = 2
Number = 3
Number = 4
Number = 5
```

In the above program control of for loop starts with the initialization of i as 1 and then the condition get checked which returns TRUE as  $1 \leq 5$ . Statement 1 will print the number as 1. Now increment statement will get executed, i will get incremented to 2 and again the condition will check. This happens 5 times until value of i remain as 5. As soon as the value of i becomes 6 condition get checked and return FALSE and control transfers out of the for loop.

Example: Program to calculate factorial of a number

```
main()
{
 int n,i,f=1;
```

```

printf("Enter a number ");
scanf("%d", &n);

for(i=1 ; i<=n ; i++)
{
 f = f * i;
}
printf("Factorial is %d", f);
}

```

Output

Enter a number 4

Factorial is 24

User enters a number which is stored in n. This loop executes n times as it goes from 1 to n. Let us suppose value entered by user is 4. So n contains 4. Loop starts with initialization of i from 1. After initialization, termination condition is evaluated, which is TRUE, thus control moves inside the code block. Initially f is containing 1 and it is multiplied by i which also contains 1. The product is finally stored in f. Now control moves to the flow part (i++). Condition is evaluated again but this time i becomes 2. Condition is again TRUE. Control again enters in code block and the product of f and i will store in f. Same process is repeated till the condition i<=n becomes FALSE.

## 5.5 Nested Loop

Using loop inside the body of loop is known as nesting of loops. Sometimes it is desirable to repeat the repetitions.

Example: Write a program to print 5 lines with each containing 10 asterisk (\*) symbols.

```

main()
{
 int i,j;
 for(i=1; i<=5; i++)
 {
 for(j=1;j<=10;j++)
 printf("*");
 printf("\n");
 }
 getch();
}

```

In the above program, we have two action statements in the outer loop. First is a for loop and second is a printf() function responsible to move cursor on the next line. These two statements will execute for 5 times. So the inner loop will work out for five times, which means printf("\*") encounters 50 times.

## 5.6 The Keyword **break**

The keyword **break** is used only in:

- Loop body
- Switch body

### break in loop body

The keyword **break** is used in loop body to terminate loop. When **break** executes, it transfers the control out of loop body without checking any loop condition.

Whenever you are not sure with when the loop should stop, you can use **break** to terminate loop.

Example:

```
main()
```

```
{
```

```
 int x, i=1;

 while(i<=5)
 {
 printf("Enter an even number");
 scanf("%d", &x);
 if(x%2==0)
 {
 printf("You Win");
 break;
 }
 i++;
 }
 if(i==6)
 printf("You lost");
}
```

In this program, **break** is used to terminate loop when user enters an even number. This program is a sort of game where user has at most 5 chances to enter correct value. If he fails to enter correct value in all five chances he would lose the game. He may win the game if he enters correct value in any of the five chances. Game must stop as soon as he enters the correct data.

Let us assume first time user enters 3. Since x contains 3, condition  $x \% 2 == 0$  is FALSE so skipped if block and increments i by 1. On its second iteration, assume user enters 6. This time condition  $x \% 2 == 0$  is TRUE, so enters in if block. User gets a message "You Win". Now

break works and it transfers the control out of while block. Where another condition is evaluated, which is false.

Try executing above program for different inputs.

### 5.7 The Keyword **continue**

The keyword continue is used only in loop body. It is used to move control at termination condition in the case of **while** and **do-while** loop. **continue** is used in for loop to transfer control at flow part (increment/decrement).

Example

```
main()
{
 int x;
 while(1)
 {
 printf("Enter an even number");
 scanf("%d", &x);
 if(x%2==1)
 continue;
 else
 {
 printf("This is the correct value");
 break;
 }
 }
}
```

This time condition of while loop is always TRUE as we wrote 1 which is a non zero number. (Non zero number is always treated as TRUE). The loop only ends at the execution of break. If the user enters an odd number condition  $x \% 2 == 1$  becomes TRUE, thus continue works. Continue transfers the control at while loop condition.

As long as user enters odd number, continue works every time, but when user enters an even number, break terminates loop.

### 5.8 switch case control Instruction

We studied earlier in decision control that how and when to use if, if-else.

They are good enough to select one from the two available options. When choices are two, we can use if-else. Even if we have multiple choices we can use multiple if-else, sometimes we use if-else ladder or nested if-else. You probably experienced that as the number of available options/choices are large, if-else becomes complex.

So when we have many choices, we can use switch case control in place of if-else.

In this section we are going to use three keywords **switch**, **case** and **default**

### Syntax of switch case

```
main()
{

 switch(expression)
 {
```

```

 case constant1 :

 case constant2 :

 case constant3 :

 ...
 default:

 }
.....
}

```

Switch case control is used when user have multiple choices. Switch transfers the control to a case written in its body depending on the value evaluated by expression in switch parenthesis. In switch body each case is post fixed by a constant. This constant could be integer or character but not real. It is also worth mentioning here that the constant in each case must be distinct.

When value of expression does not match with any case constant then control moves on default segment.

Another important point to notice that once control moves from switch to appropriate case, it not just execute statement written in that case but also execute all the statements written in following cases including default, but not cases written above. If you are concern with only one case execution, use break keyword as the last statement of every case

## 5.9 Menu driven Programming

Switch case control is very useful in menu driven program. The following example illustrates menu driven program. It is also use break to terminate switch

```

main()
{
 int a, b, result, ch;

 while(1) // condition is always true, thus an infinite
loop
 {
 clrscr();
 printf("\n1. Addition");
 printf("\n2. Subtraction");
 printf("\n3. Multiplication");
 printf("\n4. Division");
 printf("\n5. Exit");
 printf("\nEnter your choice");
 scanf("%d", &ch);

switch(ch)
{
 case 1:
 printf("Enter two numbers");
 scanf("%d%d", &a, &b);
 result=a+b;
 printf("Sum is %d", result);
 }
}

```

```

 break; //it is used to move control out of switch
body
 case 2:
 printf("Enter two numbers");
 scanf("%d%d", &a, &b);
 result=a-b;
 printf("Difference is %d", result);
 break; //it is used to move control out of switch
body
 case 3:
 printf("Enter two numbers");
 scanf("%d%d", &a, &b);
 result=a*b;
 printf("Product is %d", result);
 break; //it is used to move control out of switch
body
 case 4:
 printf("Enter two numbers");
 scanf("%d%d", &a, &b);
 result=a/b;
 printf("Quotient is %d", result);

 break; //it is used to move control out of switch
body
 case 5:
 exit(0);
 default:
 printf("Invalid Entry");
}
getch();
}
}

```

### **Explanation:**

There are few things to discuss about above program.

- 1) Notice while loop which executes infinite times till you select 5 from the menu.
- 2) In case 5, we use a predefined function exit. The job of this function is to terminate program. Argument 0 in the function depicts the normal termination. Argument could be 1 passed in the function in the case of abnormal termination.
- 3) The keyword break is used in each case as it transfers the control outside switch body.
- 4) Whenever wrong selection from menu (other than value from 1 to 5) switch moves the control to default segment.
- 5) No need to put keyword break after default statements as it is already at the end of switch body.

Rest you can understand the flow of program by executing it.

### **5.10 goto controlInstruction**

The goto statement is used for unconditional jump from one part of the program to another part of the program. It is always suggested not to use goto statement as this reduces the

readability of the program. Use of goto statement is considered as poor programming approach.

#### Example

```
main()
{
 int cm;
 printf("Enter length in centimeters");
 scanf("%d", &cm);
 if(cm<100)
 goto label;
 cm=cm%100;
 printf("Wrong input is trimed");

label:
 printf("cm=%d", cm);
}
```

Explanation:

- 1) goto is a keyword that moves control to a location in the program mentioned by location name ‘label’.
- 2) If value of cm is more than or equal to 100, it should be trimmed. For example if user enters 435, it becomes 35, 4 should be trimmed out.
- 3) Try executing this program for different inputs.

#### More points about goto

- 1) goto can move control in forward as well as backward in the program.
- 2) goto works only within the same function body.
- 3) Labels are names but no need to declare them as variables.

#### Check Yourself:

#### Objective Questions

- 1) What is the output of the following code?

```
main()
{
 int a=0,k=1;
 while(a<10)
 {
 k++;
 a++;
 }
 printf("%d %d",a,k);
}
```

- a. 9 10                      b. 10 10  
c. 10 11                      d. 9 9

2) What is the output of the following program?

```
main()
{
 int a=0,k=10;
 while(a<k)
 {
 k=k-a;
 a++;
 }
 printf("%d %d",a,k);
}
```

a. 3 7                      b. 4 4  
c. 9 10                      d. 4 10

3) Find the output of the following program

```
main()
{
 int k=1,x=3,y=4,a;
 for(a=y;a>0;a--)
 {
 k=k*x;
 }
 printf("%d",k);
}
```

a. 81                      b. 729  
c. 27                      d. 0

4) Find the output of the following program

```
main()
{
 int a,b=5,count=0;
 for(a=4;a>0;a--)
 {
 for(;b<10;b++)
 if((a+b)%2==0)
 count++;
 }
 printf("%d",count);
}
```

a. Error                      b. 2  
c. 8                              d. 5

5) Find the output of the following program

```
main()
{
```

```

int a=0,k=10;
while(1)
{
 k=k+a; a=k-a; k=k-a;
 if(a<k) break;
 else a++; k-=2;
}
printf("%d %d",a,k);
}

```

a. -2 11                      b. 4 4  
c. 0 11                      d. 2 11

- 6) Find the output of the following program

```

main()
{
 int a=0;
 while(a<10)
 {
 if(a%3==0) {a++; continue;}
 if(a%4!=0) {a+=2; }
 if(a%5!=0) { a++; break;}
 }
 printf("%d",a);
}

```

a. 0                              b. 2  
c. 4                              d. 5

- 7) Find the output of the following program

```

main()
{
 int a=3;
 switch(a)
 { case 1: printf("Hello");
 case 2: printf("Hilo");
 case 3: printf("Mat Hilo");
 default: printf("Khub Hilo");
 }
}

```

a. Mat Hilo                      b. Mat Hilo Khub Hilo  
c. Khub Hilo                      d. Error

- 8) Find the output of the following program

```

main()
{
 int a=2;
}

```

```

switch(a)
{
 case 1: printf("Hello"); break;
 case 2: printf("Hilo"); continue;
 case 3: printf("Mat hilo"); break;
 default: printf("Khub Hilo");
}

```

a. Blank Screen                      b. Mat Hilo  
 c. Khub Hilo                        d. Error

9) Find the output of the following program

```

main()
{
 int a=2;
 while(a>=-5 && a<=5) {
 switch(a)
 {
 case -1: a+=2;
 case 2: a-=3; break;
 case 3: a+=4; break;
 default: a-=5;
 }
 printf("%d ",a);
 }

```

a. -1 2 -7                      b. -1 -2 -7  
 c. 1 -2 -7                        d. None of these

10) Find the output of the following program

```

main()
{
 int a;
 for(a=2;a<=10;a++) {
 switch(a%3)
 {
 case 0: a+=2;
 case 1: a-=3; break;
 case 2: a+=4; break;
 default: a-=5;
 }
 printf("%d ",a);
 }

```

a. 6 4 9 7 12                      b. 6 4 9 7  
 c. 6 4 9 7 10                     d. Error

**Write the following programs**

- 1) Write a program to print first 10 natural numbers.
- 2) Write a program to print first 10 natural numbers in reverse order.
- 3) Write a program to print first 10 odd natural numbers
- 4) Write a program to print first 10 even natural numbers
- 5) Write a program to print first N natural numbers. Value of N is given by user.
- 6) Write a program to print first N odd natural numbers in reverse order. Value of N is given by user.
- 7) Write a program to calculate sum of first N natural numbers. Value of N is given by user.
- 8) Write a program to calculate product of first N natural numbers. Value of N is given by user.
- 9) Write a program to calculate factorial of a number.
- 10) Write a program to calculate  $x^y$ . Values of x and y are given by user.
- 11) Write a program to count number of digits in a given number.
- 12) Write a program to calculate sum of the digits of a given number.
- 13) Write a program to reverse a number.
- 14) Write a program to check whether the two given numbers are reverse of each other or not.
- 15) Write a program to check whether a given number is Prime or not.
- 16) Write a program to print all prime numbers between two given numbers.
- 17) Write a program to print all prime numbers in first 1000 natural numbers.
- 18) Write a program to find LCM of two numbers.
- 19) Write a program to find HCF of two numbers.
- 20) Write a program to print N numbers of Fibonacci series. Value of N is given by user.
- 21) Write a menu driven program for addition, subtraction, multiplication and division  
(Hint: use switch case)

### **Explore and Answer the following**

- 1) Write a program to print all prime factors of a given number.
- 2) Write a program to move your name with arrow keys
- 3) Write a program to move your name horizontally until user presses any key.
- 4) Write a program to print all ASCII characters and corresponding ASCII codes.
- 5) Write a menu driven program to display user's name in selected colour. Following colours should be in the choice list: RED, GREEN, BLUE, YELLOW. Repeat the menu till he choose exit from the choice list.

# Chapter 6: Functions and Storage Classes

## Content

- 6.1 Functions
- 6.2 Modularization
- 6.3 Technical Terms
- 6.4 Function Types
- 6.5 Defining Function
- 6.6 Recursion
- 6.7 Storage Classes
- 6.8 Automatic Storage Class
- 6.9 Register Storage Class
- 6.10 Static Storage Class
- 6.11 External Storage Class

### **6.1 Functions**

Functions in C language are small blocks of code, design to achieve a particular task. The main program need of such task arises. So by dividing a large programme into multiple functions we get following advantages as follows

1. The redundancy of statements can be reduced since instead of repeating the same set of lines again and again, we can write down the same code in a single line of statement.
2. The debugging of program becomes easier if it is divided into multiple function. This is because we do not find errors into the entire code rather we just have to look for them in a smalll portion of our program i.e. inside the function body. Moreover any correction done inside the function are automatically reflected at every place where the function has been called.

3. We can also create our own header files containing our own functions which we can use in any program as and when needed by including the header file .

## 6.2 Modularization

Modular programming is a software design technique that increases the extent to which software is composed of separate, interchangeable components, called **modules** by breaking down program functions into modules, each of which accomplishes one function and contains everything necessary to accomplish this

Suppose a C program is designed to perform a task. And say this task can be divided into few subtasks, each of which is independent functionality. Each subtask can be coded in a separate block known as function.

Program Execution always starts from main(). main can then call other function. Any function can call other function. This call is an invitation to that function to perform its subtask.

## 6.3 Technical terms

- **Calling Function:** Function who call another function
- **Called Function :** Function who is called by another function
- **Function call:** Action of calling a function

You have been using few functions from the starting of programming. For example printf(), scanf(), clrscr() and getch(). These are all functions and you often call them to perform task for which they are designed.

There are many more such functions. They are called predefined functions. Their code is kept in library files. These files are comprehensively called built-in C library

## 6.4 Function Types

A) **Predefined Functions OR Library Functions:** These are the functions which are pre defined in library file. We do not need to create them. We can simply use them by including corresponding header file in our program.

B) **User Defined Functions:** When user want to create a function which is not available in the library of C. These functions are known as User defined functions.

### Step Required To Develop A Function:

To develop a User Defined Function we need following steps.

1. Function prototype/Function declaration

It is the first step of designing a function to tell the compiler that what is our function name and what will be its return type and what type of values it will take.

Syntax of function declaration :

<return type> <function name>(<list of arguments>);

Components of function declaration:

1.Return type : A function can return a value to the place from its being called using return keyword.

2. Function Name: We must give a proper name to a function and remember that it should not be any reserve keyword of c nor any standard function.
3. Function parameters/Function arguments: the values which we pass inside a fuction to operate on it.

Eg. float average(int,int,int)

2. Declaration of some standard functions:
3. void clrscr();
4. double sqrt(double);
5. int strcmp(char[],char[]);
6. double pow(double,double);
7. Using Delay()/sleep(): declared in <dosh.h>

## 2.Function call

## 3. Function Definition or function body

### Difference between header file and Library file

| <b>Header Files</b>                                                                         | <b>Library Files</b>                                                                            |
|---------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| It contains declaration of pre defined functions                                            | It contains function definition                                                                 |
| They have .h file extensions                                                                | These have .lib extensions                                                                      |
| Header files are located Inside sub directory Include                                       | These are located inside Lib folder                                                             |
| Header files are included in our programme by pre-processor(i.e. #)                         | It is the linker which links the library files with our programme while execution               |
| The pre-processor executes the statements and copies the entire coding inside header files. | All the functions we have called in our programme get linked with their actual bodies by linker |
| These files are in human readable format                                                    | These are in binary format                                                                      |

Note:

1. C doesn't support function overloading.
2. a function may accept none or multiple values but can return at most single value .

### Types of Functions:

On the basis of return type and parameters function has 4 types.

- 1.Take something, Return Something.  
pow(),sqrt(),strlen(),strcmp() etc.
2. .Take something, Return nothing.  
fflush(),delay(),sleep(),textcolor(),gotoxy()

```
#include<stdio.h>
#include<conio.h>
```

```

void main()
{
int i;
clrscr();
for(i=1;i<=10;i++)
{
printf("\n",i);
sleep(1)/*delay(1000) here delay and sleep pause execution for a while delay takes time in
milli seconds while sleeps takes in second*/
getch();
}
Using textcolor and cprintf
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
textcolor(YELLOW);
cprintf(" Sanjeev");
getch();
}

```

3. Take nothing, Return something.

```
getch();
```

4. Take nothing ,Return nothing.

```
clrscr();
```

### **User Define Function Example:**

Function take something and return something

```

#include<stdio.h>
#include<conio.h>
int sum(int,int) /*int,int are signature in prototype declaration*/
void main()
{
```

```

int n, x,y;
clrscr();
printf("enter two integers");
scanf("%d %d",&x,&y);
n=sum(x,y) /*x,y are actual parameters* and this is function call
printf("sum of x and y is=%d",n)
getch();
}
/*function definition*/
int sum(int p,int q)//p and q are formal parameters. Values of actual args are copied to formal
args.
{
return(p+q);
}

```

Exaple of funtion take something but return nothing

```

#include<stdio.h>
#include<conio.h>
void sum(int,int) /*int,int are signature in prototype declaration*/
/*void is special data tpye used when function take nothing to return nothing*/
void main()
{
int n, x,y;
clrscr();
printf("enter two integers");
scanf("%d %d",&x,&y);
n=sum(x,y) /*x,y are actual parameters* and this is function called/
printf("sum of x and y is=%d",n)
getch();
}
/*function definition*/
void sum(int p,int q)//P and Q are formal parameters
{
printf("Sum is=%d",p+q);
}

```

Exaple of funtion take nothing but return something

```

#include<stdio.h>
#include<conio.h>
int sum(void) /*int,int are signature in prototype declaration*/
/*void is special data tpye used when function take nothing to return nothing*/
void main()

```

```

{
int n;
clrscr();
n=sum() /*x,y are actual parameters* and this is function called/
printf("sum of x and y is=%d",n)
getch();
}
/*function definition*/
void sum(int p,int q)//P and Q are formal parameters
{
int x,y;
printf("enter two integers");
scanf("%d %d",&x,&y);

return(p+q);
}

```

Exaple of funtion take noththing and return nothing

```

#include<stdio.h>
#include<conio.h>
void sum(void) /*int,int are signature in prototype declaration*/
/*void is special data tpye used when function take nothing to return nothing*/
void main()
{
clrscr()
getch();
}
/*function definition*/
void sum(void)//P and Q are formal parameters
{
int n, x,y;
clrscr();
printf("enter two integers");
scanf("%d %d",&x,&y);
printf("Sum is=%d",x+y);
}

```

## **10 Basic Questions**

1) Can we call a function several times in a single program?

Ans: Yes.

2) Can user define function become a calling function?

Ans: Yes.

3) Can we pass values through functions?

Ans: Yes, it is called “Call by value”.

4) Can we have a c program without main function.

Ans: No

5) Is there any limit of total number of functions that can ‘c’ programs have?

Ans: No

6) Can two functions have same name with different definitions?

Ans: No

7) Who calls main?

Ans: An operating system.

8) Can a function call main() ?

Ans: Yes

9) Can a function call itself ?

Ans: Yes, it is called recursion.

10) Why write separate functions at all? Why not squeeze the entire logic into one function main()?

Ans:

1. Writing functions avoids rewriting the same code over and over.
2. It provides strong readability.
3. It makes program easy to debug.
4. It becomes easy to modify.
5. Better memory utilization

## **6.5 Defining function**

There are four ways to define a function:

- 1) Takes nothing return nothing
- 2) Takes something returns nothing
- 3) Takes nothing returns something
- 4) Takes something and returns something

### **1) Takes nothing returns nothing**

main()

{

```
void sum(void);
clrscr();
sum();
```

```

 getch();
 }
void sum()
{
 int a,b,c;
 printf("Enter two numbers:-");
 scanf("%d%d", &a, &b);
 c=a+b;
 printf("Sum is %d", c);
}

```

### **Explanation:**

In this program we wrote two functions, one is sum() and second is main().

It is not necessary to define sum() before main(). There would be no difference in, what order of function definition is positioned in your program. This is completely a programmer choice. Whatever may the order of function definition, program execution always starts from main().

Notice the first line of main() function (void sum(void);). This is called declaration of function. Declaration of function tells the compiler about return type, function name and argument type. Since we are not returning any value we mentioned so by void. Function name is sum. Since there is no argument we mentioned it so by using keyword void in the parenthesis. We will discuss more about function declaration later in this chapter.

Second line in the main function is a call to a predefined function clrscr(). Since clrscr() function defined in such a way that it takes nothing, we left its parenthesis empty.

Next line is a call to a function sum(). This function takes nothing, so the parenthesis is empty. Due to call to function sum(), control moves to the definition of function sum(). In sum(), three int type variables are declared. After call to printf() and scanf(), c is assigned a value which is an addition of value stored in a and b. Lastly the call to printf() displays the result.

When all the statements of function sum() worked out, control returns back from where sum() was called. Hence control comes back to main() and the last function getch() execute.

### **2) Takes something returns nothing**

```

main()
{
 void sum(int, int);
 int a,b;
 clrscr();
 printf("Enter two numbers:-");
 scanf("%d%d", &a, &b);
 sum(a, b);
 getch();
}
void sum(int x, int y)

```

```

{
 int c;
 c=x+y;
 printf("Sum is %d",c);
}

```

**Explanation:** This time we passed two int type values during call to function sum(a,b). these values are called **actual arguments** or parameters. Notice the subsequent change in function declaration and definition. Values of a and b are collected in x and y. Variable x and y are called **formal arguments**.

It is important to note that any function can access only those variables that are declared in their body. So the scope of variable is limited to the function in which it is declared.

### 3) Takes nothing returns something

```

main()
{
 int sum(void);
 int s;
 clrscr();
 s=sum();
 printf("Sum is %d",s);
 getch();
}

int sum()
{
 int a,b,c;
 printf("Enter two numbers:-");
 scanf("%d%d",&a,&b);
 c=a+b;
 return(c)
}

```

**Explanation:** In this example we do not pass anything but function sum returns an int type value, which is addition of two numbers. Notice the change in return type in function declaration and definition part. Also notice how we collect the return value in function main. Whatever sum() returns, it goes back at the same place from where a function is called. It is then assigned to s.

The keyword return is used to return value of any type. We can only return single value using return.

As soon as return is executed control comes back to the calling function, thus it is meaningless to expect any statement could execute after return in function sum().

### 4) Takes something returns something

```

main()
{
 int sum(int,int);
 int s;

```

```

clrscr();
printf("Enter two numbers:-");
scanf("%d%d", &a, &b);
s=sum(a,b);
printf("Sum is %d", s);
getch();
}

int sum(int x,int y)
{
int c;
c=x+y;
return(c);
}

```

**Explanation:** This time function sum() is called by passing two int values and addition is returned back to calling function.

### Another example:

#### Program to calculate area of a circle.

```

float area(int);
main()
{
 int r;
 float a;
 clrscr();
 printf("Enter radius of circle:-");
 scanf("%d", &r);
 a=area(r);
 printf("Area of circle is %f ",a);

 getch();
}
float area(int r)
{
 float A;
 A=3.14*r*r;
 return(A);
}

```

### Function prototype

Function prototype also known as function declaration. Although you have already seen how functions has to be declared. Here we would like to throw some light on function declaration.

- 1) In C language, function declaration is recommended but not mandatory. Some of the compiler allow programmer to use function without declaration, but we recommend to declare all functions used in your program. It is so because subsequent languages including C++ are very strict about function declaration.
- 2) Function prototype has general form:

**Return type function name ( argument type);**

- 3) All predefined functions are already declared and their declaration is kept in some header file, all you need to include that header file. For example printf() and scanf() functions are declared in stdio.h. Similarly, clrscr() and getch() are declared in conio.h

**Few Points to be noted regarding function:**

**Consider below sample program:**

```
void main()
{
 int add(int,int);
 int a, b, c;
 clrscr();
 a=10;b=20;
 c=add(a,b);
 printf("\n Addition of %d and %d results in %d",a,b,c);
}
int add(int x, int y)
{
 int z;
 z=x+y;
 return(z);
}
```

## 6.6 Recursion

Function calling itself is called recursion.

Example:

```
int fun(int);
void main()
{
 int x, k=3;
 x = fun (k);
 printf("%d", x);
}
int fun (int a)
{
 int f;
 if (a == 1) return(1);
 f = 2 + fun (a - 1);
 return(f);
}
```

**Output:**

5

**Explanation:**

Operating system calls **main()** function. Memory for **x** and **k** are being allotted. **k** is assigned with 3. Now **fun()** is called by **main()** and passes value of **k**(called by value). Memory for the function **fun(3)** is allotted, in which there are two variables **f** and **a**. **a** contains 3.

Now condition **a==1** is false so control moves on **f=2+fun(a-1)**. Here, again make a call to function **fun()**. **fun()** is calling **fun()**, this is called recursion.

Remember this time a new and separate memory is being allotted, call it **fun(2)**. It also uses the same definition but its variable **a** contains 2. The condition **a==1** is again false. And again control moves to **f=2+fun(a-1)**. Again function **fun(1)** is called from **fun(2)** and recursion continues. Again separate memory is allocated for **fun(1)**. This time **a** would contain 1 and the condition **a==1** is true and return(1) to the calling function that is **fun(2)**. You can easily understand these successive returns to the calling functions.

At last **x** would contain 5

## 6.7 Storage Classes

Data type declaration instruction depicts following characteristics of a variable:

- 1) Name of the variable
- 2) Size of memory block
- 3) Type of the content that it can store

For example

```
main()
{
 int x;
 ...
}
```

Here we can easily understand that the variable name is ‘**x**’. Size it takes is two bytes. Type of the content it can store is integer. So we can say data type declaration instruction provides three basic properties of variable. One can learn these properties by just examine data type declaration instruction.

Apart from these three properties of a variable, there are four more properties of a variable.

A variable in the program belongs to one of the four storage classes. These storage classes talk about following four properties of a variable:

- Storage
- Default value
- Scope
- Life

Following are the four storage classes:

- 1) Automatic Storage Class
- 2) Register Storage Class
- 3) Static Storage Class
- 4) External Storage Class

## 6.8 Automatic Storage class

- Keyword used is auto
- Storage in main memory (RAM)
- Default value is garbage
- Scope is limited to the block in which it is declared.
- Life retains till the control moves in the block in which it is declared

Example

```
main()
{
 auto int x = 2;
 printf("%d ", x);
 {
 auto int x = 5;
 printf("%d ", x);
 }
 printf("%d ", x);
}
```

**Output is 2 5 2**

Some strange things, like declaration of x two times and declaration instruction after printf(). You remember the data type declaration instruction is possible only if it occur before any other action statement. But this rule has a scope within the block. We are using two blocks. Second declaration is the first statement in the inner block hence it is perfectly legal. Scope and life of second x is limited to the block hence cannot be accessed outside.

## 6.9 Register Storage Class

- Keyword used is register
- Storage in register
- Default value is garbage
- Scope is limited to the block in which it is declared.
- Life retains till the control moves in the block in which it is declared.

Register storage is a request only, allotment of register to the variable is depend on the availability of register which is monitored by operating system. Sometimes we feel a particular variable is used frequently, in such cases we do request to the operating system to hold its data in the processor register so the program execution becomes faster.

## 6.10 Static Storage Class

- Keyword used is static
- Storage in main memory
- Default value is 0 (zero).
- Scope is limited to the block in which it is declared.
- Life retains till the execution of program.

**Example:**

```
main()

{
 inc();
 inc();
}

inc()

{
 static int x;
 printf ("%d", ++x);
}
```

**Output:**

12

**Explanation:** In this program, x is a static variable in function inc(). Static variables are created only once in a program but its life time is whole program. Scope of static variable is limited to the block in which it is declared, thus we cannot access variable ‘x’ from outside the inc() function. Since the variable remains alive in memory in subsequent function call of inc(), any change in variable ‘x’ is available for next call to function inc().

### 6.11 External Storage Class:

- Keyword used is extern
- Storage in main memory
- Default value is 0 (zero).
- Scope is global.
- Life retains till the execution of program.

**Example:**

```
int x=5; //global variable

main()

{
 printf ("%d", x);

 fun();
}

int fun()
```

```
{
 printf("%d", x);
}
```

## Output

55

External variables are declared outside all functions. They are also known as global variables. All function in the program can access global variable. There will be a single copy of global variable, shared between all functions.

If any function declared a local variable with the same name in its body, local variables get preference over global variable.

The keyword extern should not be used during declaration of global variable. It should be used to inform compiler if we are using any global variable in the block.

Let us take an example:

```
main()
{
 printf("%d", x);
 fun();
}
int x=5; //global variable
int fun()
{
 printf("%d", x);
}
```

In the above code, global variable is declared after main function. Compiler will yield an error message as undefined symbol x in main function. This happens due to early encounter of action statement using variable x and the variable declaration is yet not visited. To overcome this issue we must mention extern statement in main().

Here is the solution:

```
main()
{
 extern int x;
 printf("%d", x);
 fun();
}
int x=5; //global variable
int fun()
```

```
{
 printf("%d", x);
}
```

## **Check Yourself:**

### **Objective Questions**

- 1) What is the output of the following code?

```
#include<stdio.h>
int fun(int,int);
main()
{
 int x=3,y=4;
 fun(x,y);
 printf("%d %d", x, y);
}
fun(int y, int x)
{
 x=x+y; y=x-y; x=x-y;
}
 a. 3 4
 b. 4 3
 c. Error
 d. 3 3
```

- 2) What is the output of the following program?

```
#include<stdio.h>
int fun(int,int);
main()
{
 int x=3,y=4;
 fun(x, y);
 printf("%d %d", x, y);
}
fun(int y, int x)
{
 x=x+y; y=x-y; x=x-y;
 return(x,y);
}
 a. 3 4
 b. 4 3
 c. Error
 d. 4 4
```

- 3) Find the output of the following program

```
#include<stdio.h>
int max(int,int);
```

```

int min(int,int);
main()
{
 int k,x=3,y=4;
k=max(max(x,y),min(y,x));
 printf("%d",k);
}
max(int x, int y) { return(x>y ? x: y); }
min(int x, int y) { return(x>y ? y: x); }
 a. 3 b. 4
 c. 5 d. 2

```

- 4) Find the output of the following program

```

#include<stdio.h>
int f1(int);
int f2(int);
main()
{
 int k,x=3;
 k=f1(f2(f1(f2(f1(x)))));
 printf("%d",k);
}
f1(int x) { return(++x); }
f2(int x) { return(x++); }
 a. 4 b. 5
 c. 6 d. 7

```

- 5) Find the output of the following program

```

#include<stdio.h>
int f1(int);
int f2(int);
main()
{
 int k,x=3;
 k=f1(x);
 printf("%d",k);
}
f1(int x) { return(f2(++x)); }
f2(int x) { return(--x); }
 a. 2 b. 3
 c. 4 d. 5

```

- 6) Find the output of the following program

```

#include<stdio.h>
int f1(int);

```

```

int f2(int);
main()
{ int x=24,y=36;
 printf("%d",abc(x,y));
}

abc(int a,int b)
{ if(a==b) return(a);
 if(a>b) if(a%b==0) return(b);
 else return(abc(a%b,b));
 else if(b%a==0) return (a);
 else return(abc(a,b%a));
}
a. 12 b. 18
c. 6 d. 4

```

- 7) Find the error in the following program

```
dummy(int a, int b)
```

```
{
 int a;
 a=20;
 return a;
}
```

- a. Missing parenthesis in return statement
- b. Function return type should be mentioned
- c. Re-declaration of variable a
- d. None

- 8) Find the error in the following program

```
#include<stdio.h>
int fun(int);
int main()
{
 int b;
 b=fun(20);
 printf("%d", b);
 return(0);
}
int fun(int a)
{
 a>20 ? return(10): return(20);
}
```

- a. Error in prototype
- b. No error
- c. The return statement cannot be used with conditional operator
- d. The function main cannot return any value

- 9) Find the output of the following program

```
#include<stdio.h>
void fun (void);
main()
{
 fun();
 fun();
 fun();
}
void fun()
{
 static int x;
 x++;
 printf("%d",x);
}
```

- a. 123                    b. 111  
c. garbage                d. Error

- 10) Find the output of the following program

```
#include<stdio.h>
main()
{
 extern int a;
 a=3;
 printf("%d ",++a); }
```

- a. 3                    b. 4  
c. 0                    d. Error

### Write following programs

- 1) Write a function to calculate area of circle. (Take something, return something)
- 2) Write a function to calculate factorial of a number. (Take Something, Return Something)
- 3) Write a function to calculate LCM of two numbers. (Take Something, Return Something)
- 4) Write a function isPrime() to check if a number is prime or not. Function should take an integer as an argument and should return 1 if the number is prime, otherwise return 0.
- 5) Write a recursive function to calculate sum of N natural numbers. (Take something return something)
- 6) Write a function to find greatest among three numbers. (Take something return something)

- 7) Write a recursive function to calculate sum of squares of first N natural numbers.  
(Take something return something)
- 8) Write a recursive function to calculate sum of cubes of first N natural numbers. (Take something return something)
- 9) Write a recursive function Fibo to print n numbers of Fibonacci series. (Take something return nothing)

### **Explore and Answer the following**

- 1) Write a program to display n rows Pascal triangle. The value of n is given by user. To accomplish the task define functions fact() to calculate factorial, combi() to calculate combination.
- 2) Write a recursive function to calculate greatest common divisor of two numbers
- 3) Write a program to run dos command. Demonstrate few of the dos commands.
- 4) Write a program to change your text color. Use predefined functions to change color of the text.
- 5) Write a program to play a number guessing game with three chances given to user, give hint on each chance as whether the secret number is greater or smaller from the guessed number. Secret number should be any number from 1 to 25.
- 6) Write a program to calculate sum of N natural numbers without using loop.
- 7) Explore the use of functions: delay(), kbhit(), gotoxy(), sound(), nosound(), rand(), randomize()
- 8) Write a program to print system date and time
- 9) Write a menu driven program to demonstrate the use of functions from math library such as, sqrt, pow, sin, log, exp, abs, etc
- 10) Write a program to implement Tower of Hanoi

## **Chapter 7: Array**

### **Content**

- 7.1 Arrays
- 7.2 Array Declaration
- 7.3 Initializing Arrays
- 7.4 Memory allocation in an array
- 7.5 Two dimensional and Multi-dimensional Arrays
- 7.6 Initialization of two dimensional Array

### **Introduction**

Till now we have been designing solution to small problems that require less number of variables to handle program data. Think about a scenario where you need to handle hundreds of variables or even more than that. In such scenario, you might be thinking about what variable names should be used, how to reduce redundant code, etc.

Assume you have to store marks of 100 students and then think about the following:

- 1) What could be your variable naming convention?
- 2) How you can efficiently write input instruction to store 100 data.
- 3) How could you easily manipulate data like adding all of them in a less complex style?

The answer to all these questions is subscript notation also known as **Arrays**.

## 7.1 Arrays

Array is a collection of similar elements. These similar elements could be all ints, floats, doubles, chars, etc. Array is also known as subscript variable.

C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. Array elements are stored in contiguous memory block.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

Properties of Array:

- 1) Indexing of an array begins from zero (0).
- 2) The variable name of array contains the base address of the memory block.
- 3) The array variable are created at the time of compilation.
- 4) The size of the array cannot be altered at runtime.

## 7.2 Array Declaration

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows

<data type> arrayName [arraySize];

This is called single dimension Array.

[] is subscript operator and arraySize is the size of an array ie. how many elements are going to be stored in an array.

```
int roll[10]/*roll is an array of 10 integers
char grade[5]/*grade is array of 5 characters
float per[15]/* per is array of 15 float values structure of an array in memory
int arr[5];
i int takes 2 bytes and size of array is 5 means total 5*2=10 , bytes will be reserved in
memory for arrray. also array elements are indexed in memory and this index starts from 0 to
array size-1
```

| Addres in memory | Arr elements | Index no |
|------------------|--------------|----------|
| 1000             | 3            | 0 arr[0] |
| 1002             | 4            | 1 arr[1] |
| 1004             | 8            | 2 arr[2] |
| 1006             | 9            | 3 arr[3] |

### 7.3 Initializing Arrays

- int a[5]={34,56,78,99,12};
- int a[5]={34,56};

Initializing an array with lesser values than its size is allowed and the remaining array blocks will contains zero.

- int a[5]={34,56,78,99,12,44,66};

We cannot initialize an array with values exceeding its size.

- int a[ ]={34,56,78,99,12};

We can declare an array without mentioning its size only when initializing array during declaration.

int a[5];

In the above declaration 5 is the size of an array or we can say an array is a collection of 5 variables.

Variables in an array do not have names. These variables can be recognized by their position number in an array known as index. Index of the very first variable is 0 (zero). So the index of last variable is 4 (according to our example). To assign value in any particular variable of an array we have to follow the syntax:

a[index]=value;

For example:

a[0]=34;

a[1]=21;

It is worth mentioning here that the value written in square bracket is size of array during declaration and it is index afterwards.

#### Example: Program to calculate average of 10 marks

```
#include<stdio.h>
#include<conio.h>
main()
{
 int i, marks[10], sum=0;
 float avg;
 clrscr();
 printf("Enter 10 numbers");
 for(i=0;i<=9;i++)
 scanf("%d",&marks[i]);
```

```

for(i=0;i<=9;i++)
 sum=sum+marks[i];
avg=sum/10.0;
printf("Average is %f",avg);
getch();
}

```

**Explanation:**

- 1) Total numbers of variables in this program are 13. Their names are i, sum, avg, marks[0], marks[1],...marks[9].
- 2) Notice the input statement, scanf() is repeated 10 times, this is possible as we can access array index with the help of variable. Observe **&marks[i]** in scanf(), here i is used for indexing. As the loop proceed value of i changes from 0 to 9. In this way we need not to write scanf() 10 times.
- 3) Data manipulation becomes also easy, we added all 10 values stored in an array putting statement **sum=sum+marks[i]** in the loop.
- 4) Since we have an array of size 10 and each of these blocks are of type int, total memory consumed for this array is 20 bytes (2 bytes for each).

## 7.4 Memory Allocation in an array

Whatever the size of an array, it always consumes memory in **contiguous manner**.

- int a[4];

In the above declaration, array size is 4 which mean an array is a collection of 4 int type variables. Thus the total memory occupied is 8 bytes.

## 7.5 Two Dimensional and Multidimensional Arrays

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

<data type> arrayName[size1][size2]...[sizeN];

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays

Example:

int a[3][4];

The above declaration says, we have an array of 3 arrays of 4 int blocks.

Array will always get memory in contiguous manner. For an ease of understanding, conceptually 2 dimensional arrays can be viewed as row column fashion. As we have three arrays each of containing four elements, we can say we have three rows (arrays) each containing 4 columns.

**Example:Program to add two matrix of order 3 x 3.**

```

#include<conio.h>
#include<stdio.h>
main()
{
 int a[3][3],b[3][3],c[3][3];
 int i,j;
 clrscr();
 printf("Enter 9 numbers for first matrix\n");
 for(i=0;i<3;i++)
 for(j=0;j<3;j++)
 scanf("%d",&a[i][j]);

 printf("Enter 9 numbers for second matrix\n");
 for(i=0;i<3;i++)
 for(j=0;j<3;j++)
 scanf("%d",&b[i][j]);

 for(i=0;i<3;i++)
 for(j=0;j<3;j++)
 c[i][j]=a[i][j]+b[i][j];

 printf("\nSum of matrix is: \n");
 for(i=0;i<3;i++)
 {
 for(j=0;j<3;j++)
 printf("%d ",c[i][j]);
 printf("\n");
 }
 getch();
}

```

**Explanation:**

- 1) Notice the declaration of arrays. We have declared three two dimensional arrays.
- 2) Observe the style of input, scanf() is repeated with nested loop.
- 3) Lastly sum of corresponding elements of two matrices are added and stored in third array.
- 4) Finally sum is displayed on the screen

## 7.6 Initialization of two dimensional Array

```

int b[2][3] = {12,65,78,45,33,21};
int b[][3] = {12,65,78,45,33,21};
int b[2][] = {12,65,78,45,33,21};

```

```
int b[][] = { {12,65,78}, {45,33,21} };
```

Last two styles are invalid and lead to compile time error.

**Note:** It is important to remember that while initializing an array it is necessary to mention the second(column) dimension, whereas the first dimension(row) is optional.

### **Check Yourself:**

## **Objective Questions:**

- 1) What is the output of the following code?

```
#include<stdio.h>
```

## main()

{

```
int x,k,a[]={1,2,3}
```

```
for(x=1;x<5;x++)
```

```
{ k=x%3; a[k]++; }
```

F



- 2) What is the output of the following program?

```
#include<stdio.h>
```

main()

{

```
int a[]={1,2,3};
```

```
printf("%d %d", a[0],1[a]);
```

}



- 3) Find the output of the following program

```
#include<stdio.h>
```

main()

{

```
int k,x,y,a[]={21,2,13};
```

```
for(x=0;x<2;x++)
```

```
for(y=x+1;y<3;y++)
```

```

if(a[x]>a[y])
{ k=a[x]; a[x]=a[y]; a[y]=k; }
printf("%d%d%d", a[0],a[1],a[2]);
}
a. 21 2 13 b. 2 21 13
c. 13 2 21 d. 2 13 21

```

- 4) Find the output of the following program

```

#include<stdio.h>
main()
{
int x,y=0,a[]={11,34,21,2,13};
for(x=0;x<5;x++)
{
if(y<a[x])
y=a[x];
printf("%d", y);
}
a. 34 b. 43
c. 2 d. 21

```

- 5) What will happen if in a C program you assign a value to an array element whose subscript exceeds the size of array?

- a. The element will be set to 0
- b. The compiler would report an error.
- c. The array size would appropriately grow
- d. The program may crash if some important data gets overwritten.

- 6) Find the output of the following program

```

#include<stdio.h>
main()
{
int x[4];
printf("%d %d", sizeof(x), sizeof(x[0]));
}
a. 8 2 b. 2 2
c. 8 8 d. Error

```

- 7) Which is the correct declaration of two dimension array

- a. int a[3][ ]={2,3,4,6,7,8,9}; b. int a[ ][ ]={2,3,4,6,7,8,9};
- c. int a[ ][4]={2,3,4,6,7,8,9}; d. All are correct

- 8) Consider the following declaration

```
int a[3][4][5]={22,34,5,6,1,0,55,4};
```

Which of the following statement is incorrect?

- a. Total memory occupied by array is 120 bytes
- b. There are 3 two dimensional arrays, each is an array of 4 arrays of 5 int blocks.
- c. Value of a[0][1][2] is 0
- d. Value of a[0][1][1] is 55

### **Write following programs**

- 1) Write a program to find greatest among 10 integers. Use array to store integers.
- 2) Write a program to find smallest among 20 integers. Use array to store integers.
- 3) Write a program which takes 10 numbers from user; store them in one dimension array. Display the sum of all even numbers. Also display the sum of all odd numbers
- 4) Write a program to sort an array of 10 integers. Use bubble sort.
- 5) Write a program to sort an array of 10 integers. Use selection sort.
- 6) Write a program to calculate value of a determinant of third order.
- 7) Write a program to add two matrices of order 3x3 each.
- 8) Write a program to multiply matrices of order 3x3.
- 9) Write a program to calculate inverse of matrix of order 3x3.
- 10) Write a menu driven program to demonstrate working of QUEUE. Define insert(), delete() and View() functions to implement basic functionalities of QUEUE.

### **Explore and Answer the following**

- 1) Write a menu driven program to demonstrate working of STACK. Define Push(), Pop() and View() functions to implement basic functionalities of STACK
- 2) Write a program to generate magic square.
- 3) Write a program to implement quick sort

## **Chapter 8: Strings**

### **Content**

- 8.1 Strings and Null character
- 8.2 About format specifier %s
- 8.3 User Input
- 8.4 String manipulation through predefined functions
- 8.5 Two dimensional char arrays

## 8.1 Strings and Null character

The string in C programming language is actually a one-dimensional array of characters which is terminated by a **null** character ‘\0’

The following declaration and initialization create a string consisting of the word “Bhopal”. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word “Bhopal”

- char city[7]={‘B’,’h’,’o’,’p’,’a’,’l’,’\0’};

Another way to initialize a string in an array is as follows:

- char city[7]=”Bhopal”;

### Example:

```
main()
{
 char city[7] = "Bhopal";
 int i=0;
 while(city[i]!='\0')
 {
 printf("%c",city[i]);
 i++;
 }
}
```

### **Explanation:**

Notice the way of initializing array. Instead of writing each character in single quotes and separating with commas, this convention is convenient to use.

Another important point to note is- there is no need to write null character at the end in this convention, compiler automatically appends null character at the end

The printf() is repeated till city[i] would not become ‘\0’ character. In each iteration it prints one character of the string.

## 8.2 About format specifier %s

### Example:

```
main()
{
 char text[6] = "Hello";
 printf("%s", &text[0]);
}
```

### **Explanation:**

Here is another program depicting the use of format specifier %s.

This is used only for strings.

Observe that in printf we write &text[0] and not text[0]. This means with %s format specifier, one should always use address of first block of array.

You can write *text* in place of &text[0], both have the same meaning. The name of the array always represents address of the very first block of an array, so text is same as &text[0].

## **8.3 User Input**

```
main()
{
 char text[20];
 printf("Enter your name");
 scanf("%s", &text[0]);
 printf("hi, %s", &text[0]);
}
```

### **Explanation:**

In this program, user can input his name as we used scanf().

Notice %s is used in scanf for string input. Again remember, if you are using %s in printf or scanf always pass address of the first block of array.

Run this code to understand the limitations of scanf().

If you input *Amir*, it will get stored in the array and null character is automatically appended by compiler.

If you input Amir Khan, only the first word of the input string will be stored in character array and nul is automatically appended.

Scanf() is not capable to input multiword string. The solution to this problem is a function gets()

### **Example:**

```
main()
{
 char text[20];
 printf("Enter your name");
 gets(&text[0]);
 printf("hi, %s", &text[0]); //or puts(&text[0]);
}
```

### **Explanation:**

No need to mention format specifier in function gets() as it is dedicated only for string input.

The function gets() is declared in stdio.h header file.

Function gets() can input only one string at a time.

As gets() can be used for input, function puts() can be used for output in place of printf().

The function printf can print multiple strings in a single statement; on the other hand the function puts can only output single string at a time.

### **Example:**

```
/*Program to calculate length of string*/
main()
{
 char text[20];
 int len=0;
 printf("Enter a string ");
 scanf("%s", &text[0]);
 while(text[len]!='\0')
 len++;
 printf("Length of string: %s is %d", &text[0], len);
}
```

### **Explanation:**

If you input Sachin length will be 6. If you input Saurabh length will be 7.

To calculate length of a given string, we take a variable len. len is initialized with 0. len is now incremented till text[len] becomes nul. Since len is now containing index of that block which contains nul. The value of len is length of the string excluding nul character.

No need to decrease len by 1 to show length at last, as indexing starts from 0.

## **8.4 String manipulation through predefined functions**

There are several predefined functions to manipulate strings. These functions are declared in string.h. We are discussing few of them:

### **1) Function name: strlen()**

#### **Usage:**

```
main()
{
 int len;
 char str[20];
 printf("Enter a string");
 gets(str);
 len=strlen(str);
```

```
 printf("Length is %d", len);
}
```

## 2) Function name: strrev()

### Usage:

```
main()
{
 char str[20];
 printf("Enter a string");
 gets(str);
 printf("You entered %s", str);
 strrev(str);
 printf("Reverse is %s", str);
}
```

## 3) Function name: strlwr()

### Usage:

```
main()
{
 char str[20];
 printf("Enter a string");
 gets(str); //sample input is SCA
 strlwr(str);
 printf("String in lower case is %s", str); //sample
output is sca
}
```

## 4) Function name:strupr()

### Usage:

```
main()
{
 char str[20];
 printf("Enter a string");
 gets(str); //sample input is sca
 strupr(str);
 printf("String in upper case is %s", str); //sample
//output is SCA
}
```

## 5) Function name: strcpy()

### Usage:

```

main()
{
char str1[20], str2[20];
printf("Enter a string");
gets(str1); //sample input is Computer
strcpy(str2,str1);
printf("String1=%s and String2 = %s",str1,str2);
}

```

**Output:**

String1=Computer and String2=Computer

**6) Function name: strcat()**

**Usage:**

```

main()
{
char str1[20], str2[40];
printf("Enter two strings");
gets(str1); //sample input is Computer
gets(str2); //sample input is Education
strcat(str2,str1);
printf("String1=%s and String2 = %s",str1,str2);
}

```

**Output:**

String1=Computer and String2=ComputerEducation

**7) Function name: strcmp()**

**Usage:**

```

main()
{
int r;
char str1[20], str2[20], str3[20];
printf("Enter three strings");
gets(str1); //sample input is Hello
gets(str2); //sample input is Hello
gets(str3); //sample input is hello
r=strcmp(str2,str1);
printf("%d",r); //output is 0
r=strcmp(str1,str3);
printf("%d",r); //Output is -32
}

```

## 8.5 Two Dimension character array

This is similar to two dimension array that we studied in previous chapter.

```

main()
{
int i;
 char str[5][10];
 printf("Enter 5 strings");

 for(i=0;i<5;i++)
 gets(str[i]);

 for(i=0;i<5;i++)
 printf("\nString %d = %s ",i+1,str[i]);
}

```

**Explanation:**

A two dimensional character array can store multiple strings. According to our example we can store 5 strings each can contain 10 characters.

Notice function gets() we passed str[i] which is equivalent to &str[i][0]. Same is the case in printf().

## **Check Yourself:**

### **Objective Questions:**

- 1) What is the output of the following code?

```

#include<stdio.h>
main()
{
 char s[]="COMPUTER";
 printf("%s",&s[1]);
}

```

- a. COMPUTER      b. OMPUTER.      c. O      d. Error

- 2) What is the output of the following program?

```

#include<stdio.h>
main()
{
 printf("%c", "dennis"[1]);
}

```

- a. ennis      b. dennis      c. Error      d. e

- 3) Find the output of the following program

```

#include<stdio.h>
main()
{
 int l;
 char s[]="Sachin";
}

```

```
char t[] = "Saurabh";
l=strcmp(s,t);
printf("%d",l);}
```

a. 18                    b. -18                    c. 17                    d. -19

- 4) Find the output of the following program

```
#include<stdio.h>
#include<string.h>
main()
{
 char str1[20] = "Hello", str2[20] = " World";
 printf("%s\n", strcpy(str2, strcat(str1, str2)));
}
```

a. Hello                    b. Hello World  
c. World                    d. WorldHello

- 5) Point out the error in the following code

```
#include<stdio.h>
main()
{
 char str[] = "Bhopal";
 char s[20];
 s=str;
 puts(s);
}
```

a. Array declaration of str missing size in square brackets  
b. Syntax error in call of function puts  
c. LValue error in the line s=str  
d. No error

- 6) Find the output of the following program

```
#include<stdio.h>
main()
{
 char p[] = "%d\n";
 p[1] = 'c';
 printf(p, 65);
}
```

a. A                    b. Ac                    c. 65                    d. 65c

- 7) Find the output of the following program

```
#include<stdio.h>
main()
{
 printf(5 + "Good Morning\n");
}
```

a. Good Morning                    b. Morning  
c. M                            d. Good



- 6) Write a program to reverse a string. (with and without strlen()).
- 7) Write a program to convert string into lower case (with and without strlwr())
- 8) Write a program to convert string into upper case (with and without strupr()).
- 9) Write a program to copy a string into another char array ( with and without strcpy()).
- 10) Write a program to concatenate a string to another string (with and without strcat())
- 11) Write a program to compare two strings (with and without strcmp()).
- 12) Write a program to perform case insensitive comparison of two strings.
- 13) Write a program that takes 5 strings from user, store them in two dimension char array, now find the number of occurrence a specific character (entered by user) in 5 entered strings

### **Explore and Answer the following:**

- 1) Write a program to search a pattern in a given string.
- 2) Write a program that takes 5 strings from user, store them in two dimension char array, now display them in dictionary order
- 3) Write a program to abbreviate the name of a person. Example Rohit Kumar Bhargav should become R. K. Bhargav
- 4) Write a program to convert digits of a number into words. [For example 234 should becomes Two Four Three]

## **Chapter 9: Pointers**

### **Content**

- 9.1 Pointers
- 9.2 Address of operator
- 9.3 Indirection operator
- 9.4 Declaration of pointer

- 9.5 Size of pointer
- 9.6 Concept of base address
- 9.7 void pointer
- 9.8 Extended Concept of pointers
- 9.9 Pointers Jargon
- 9.10 Pointers arithmetic
- 9.11 Application of pointers
- 9.12 Call by reference
- 9.13 Pointer and Arrays
- 9.14 Pointers and strings
- 9.15 Array of pointers
- 9.16 Dangling Pointers
- 9.17 Const pointer and pointer to const

## **9.1 Pointers**

A pointer in C is a special variable used to hold the address of other variables. In single word we can say that we can't hold address of a other variable into ordinary variables

i.e. int a=10;

int b=&a

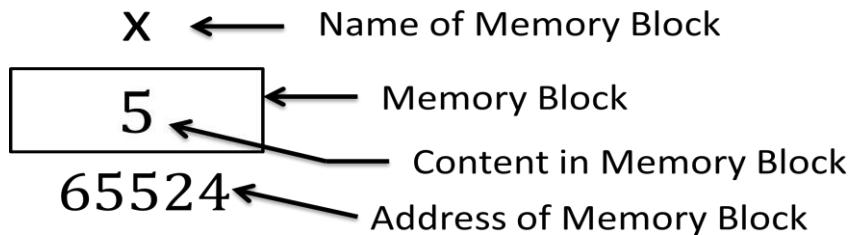
doing this is wrong .

So we need a special type of variable known as pointer.

Consider the following statement

```
int x=5;
```

The above declaration statement tells compiler about the name of memory block (variable x), amount of memory occupied by the block (2 bytes) and type of content (integer constant 5).



Every byte has a logical reference number known as address of variable. In the above example address of x is 65524. Though this address number could be anything and unpredictable to us, it is always in the range 0 to 65535.

Whatever would be the address of our variable following things are important:

- Addresses are always in the range 0 to 65535.
- Every byte will have some address number
- Address number is always an integer, even if the variable is not of type int.
- We cannot decide address number of any variable.
- Addresses are also known as references.

## 9.2 Address of Operator (&)

You are already use to of address of operator as you have been using it in `scanf()` function. Now let us examine its characteristics:

- & is called address of operator, also known as referencing operator.
- It is a unary operator, thus requires only one operand.
- Operand must be a variable.
- & returns the reference number of variable specified as the operand.

Example

```
main()
{
 int x=5;
 printf("x=%d", x);
 printf("\nAddress of x =%u", &x);
}
```

### Output

x=5  
Address of x =65524

Explanation:

We can print address of a variable by using address of operator. First printf() is a usual statement printing value of the variable. Second printf() is used to print address of variable x. Since %d can print integers comes in the range -32768 to 32767, so we are using %u to match the range. %u can display integers in the range from 0 to 65535.

### 9.3 Indirection Operator (\*)

Another important operator is indirection operator (\*).

- Indirection operator is also known as ‘value at’ operator or ‘dereferencing operator’.
- It is also a unary operator that is single operand is needed.
- Operand must be a variable address.
- It gives a representation to the block whose address is specified in the operand.

#### Example

```
main()
{
 int x=5;
 printf("x=%d", x);
 printf("\nAddress of x =%u", &x);
 printf("\nValue at address of x = %d", *&x);
}
```

#### Output

x=5  
Address of x =65524  
Value at address of x = 5

#### Explanation:

In the last printf() value at 65524 gets printed. Since &x represents address of x that is 65524, so the operand of indirection operator (\*) is 65524. As a result \*&x represents block x.

#### Important point to note:

We cannot assign anything to block by just writing address to the left of assignment operator.

```
&x=25; //Error
```

&x is not a variable. It is a way to represent address of variable x.

`&x` in itself is a constant and we cannot assign anything to constant.

#### 9.4 Declaration of Pointers

However we cannot assign anything to `&x`, we can assign address of `x` (`&x`) into some other variable. Let's say variable is `j` then we can write

```
j = &x; // Valid statement
```

This is the correct way as variable is in the left side of assignment operator.

Here `j` is a special variable, it contains address of another variable, it must be declared in a special way.

Syntax:

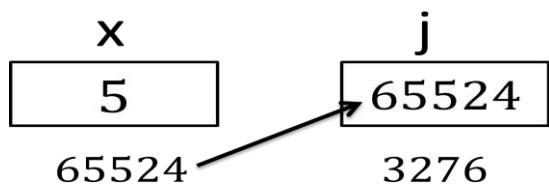
```
<data type> *<ptr_name>;
```

```
int *j;
```

Asterisk (\*) symbol before `j` in the declaration statement tells the compiler that this is a special variable meant only to store address of another variable. This special variable is called pointer.

#### Example

```
main()
{
 int i = 5, *j;
 j = &i;
 printf("\n%u", &i); //Address of i
 printf("\n%u", j); // content of j
 printf("\n%u", &j); // Address of j
 printf("\n%d", *j); //Value at address of j
 printf("\n%d", i); //content of i
 printf("\n%d", *(&i)); //Value at address of i
 printf("\n%d", *j); // Value at address in j
}
```



Output:

```
65524
65524
3276
65524
5
5
5
```

## 9.5 Size of pointer

Since pointer contains only address of another variable, they consume 2 bytes in memory. Whatever is the type of pointer, it consumes 2 bytes in memory.

### Wild pointer

When pointer is declared without initialization, it contains garbage value. This garbage is interpreted as address because it is stored in pointer variable. Obviously this garbage represents invalid address as we haven't made any reservation to that memory location. Such pointers are called un-initialized pointers or **wild pointers**.

### An important point to understand

```
float *s;
```

Above declaration does not mean that 's' is going to occupy 4 bytes and would store floating point value. It actually means 's' is going to contain address of floating point value.

's' will store an address of some float block which contains floating point value.

We can call such pointer as float pointer.

## 9.6 Concept of Base Address

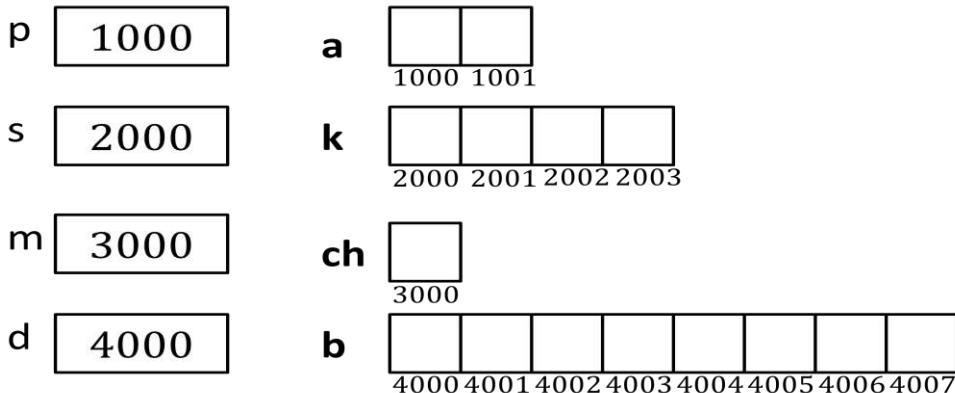
Address of the first byte of any memory block is known as base address. Size of variable depends on its type. Each byte will get address. It is possible to have multiple address of a variable, as variable can be of multiple bytes.

Consider the following declarations:

```
int *p, a;
float *s, k;
char *m, ch;
double *d, b;
p=&a;
s=&k;
m=&ch;
```

d=&b;

Now we have four pointers, each of different type. Each of the pointers consumes 2 bytes in memory. p contains address of int block 'a', s contains address of float block 'k', m contains address of char block 'ch' and d contains address of double block 'b'.



It is important to note that operating system allocates address to every byte. In our example 'a' is a variable of type int whose address is 1000 (assume), but this is the address of first byte of int block. 1001 is the address of the next byte. Similarly 2000 is the address of first byte of float block. Since float block is of 4 bytes, each byte has different address in a sequential manner.

Address of the first byte of the block of any type is known as base address of that block.

Pointer variable can store one address at a time and it is always base address. To access any block through pointer requires address of remaining bytes of the block. This is possible because of two reasons. First, addresses of bytes of a block are always in a sequence. Second, type of pointer that stores base address.

If the pointer is declared as float pointer (consider 's' in above example), it is aware of 3 more addresses to be accessed along with base address. Similarly, if the pointer is declared as double pointer (consider 'd' in above example), it is aware of 7 more addresses to be accessed along with base address.

## 9.7 void Pointer

Pointers declared with keyword void are void pointers or **generic pointers**. Since type of pointer is void, pointer can not access block, whose address it holds, by just de-referencing pointer. As we studied in the above section, pointers only hold base address and type of pointer decides how many more bytes need to be accessed. If the type of pointer is void then we can not know how many more bytes need to be accessed. Every time when we access block through pointer, typecasting is useful.

Example:

```
main()
{
```

```

void *p;
int x=3;
float k=3.14;
p=&x;
printf("%d", *((int *)p));
p=&k;
printf("%f", *((float *)p));
}

```

## 9.8 Extended Concept of Pointers

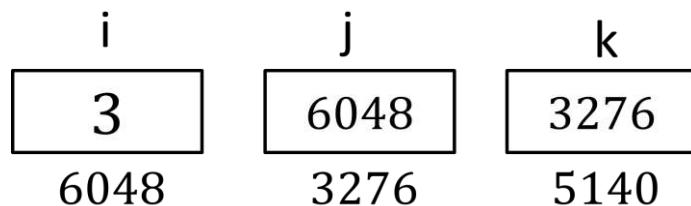
Pointer is a variable which contains address of another variable. Now this variable itself could be another pointer. Thus we now have a pointer, which contains another pointer address

Consider the following example

```

main()
{
 int i = 3, *j, **k;
 j=&i;
 k=&j;
 printf("%u %u %u\n", &i, j, *k);
 printf("%u %u %u", &j, k, &k);
 printf("\n%d %d %d", i, *j, **k);
}

```



Output is:

6048 6048 6048

3276 3276 5140

3 3 3

## 9.9 Pointer jargons

```
int i; // i is an int
int *j; // j is a pointer to an int
int **k; // k is a pointer to a pointer to an int
```

## 9.10 Pointers Arithmetic

Although C language is not very strict about compatibility issues between various type of data, but programmer has to take special care about type conversions. It is also needful in forthcoming programming languages.

**Rule 1:** Take care for the compatibility

Example

```
main()
{
 int i=3;
 char *j;
 j = &i; //Error as incompatible assignment
 printf("%d %u", i, &i);
}
```

Error: cannot convert ‘int\*’ to ‘char\*’

(Error message depends on compiler most of the C compilers do not show any error)

Example

```
main()
{
 long int i=3;
 int *j;
 j = &i; //Error as incompatible assignment
 printf("%d %u", i, &i);
```

```
}
```

Error: cannot convert 'long\*' to 'int\*'

Example

```
main()
{
 int k, i=3, *j;
 j = &i;
 k = j; //Error as incompatible assignment
 printf("%u", k);
}
```

Error: cannot convert 'int\*' to 'int'

Example

```
main()
{
 int k, i=3, *j;
 j = &i;
 k = *j;
 printf("%d", k);
}
```

The output is

3

Example

```
main()
{
 int k, i=3, *j;
 j = &i;
 k = *j + *j;
 printf("%d", k);
}
```

The output is

6

In this program \*j would mean ‘value at address contained in j’ and it is 3.

**Rule 2:** We cannot add two addresses.

Example

```
main()
{
 int *k, i=3, *j;
 j = &i;
 k = j + j;
 printf("%d", k);
}
```

Error: Invalid pointer addition.

Similarly, we cannot multiply, divide two addresses.

**Rule 3:** We cannot multiply scalar value to an address. Similarly we cannot divide an address with scalar value

**Rule 4:** Adding 1 to the pointer gives address of the immediately next block of the same type. Similarly subtracting 1 from the pointer gives address of the previous block of the same type.

### 9.11 Application of pointers

Pointers are very useful concept as it does miraculous things that we can never imagine without pointers. Pointers are memory references and capable to make changes in the memory location.

Recall the concept of functions where one function cannot access variables of the calling function. Though, we can pass values of the variables of ‘calling function’ to the ‘called function’ and called function now can use values received in formal arguments but cannot make any change in actual arguments.

If we pass address of the variables of calling functions in place of values of the variables, then called function can receive variable addresses in pointers and can access actual arguments via pointers.

Another concept was, we cannot return more than one value from a function.

When we have addresses of variables of calling function, we can assign any value (or result of some calculation) to actual arguments via pointers, which resolves the need to returning multiple values.

### 9.12 Call by reference

When a function is called by passing addresses of variables, it is known as call by reference.

Function can only access its own memory and cannot access variables of other functions but if we pass address of variables during function call, we actually give power to the ‘called function’ to access variables of ‘calling function’ via addresses of variables

Example:

```
#include<stdio.h>
#include<conio.h>
void swap(int *, int*);
main()
{
 int a,b;
 clrscr();
 printf("Enter two numbers\n");
 scanf("%d%d", &a, &b);
 swap(&a, &b);
 printf("a=%d b=%d", a, b);
 getch();
}
void swap(int *x, int *y)
{
 int temp;
 temp=*x;
 *x=*y;
 *y=temp;
}
```

Output:

Enter two numbers

10

20

a=20 b=10

Explanation:

- In the above program, two variables (a and b) are declared in main() function. They can be accessed from main() function only by their names.
- Let us assume user enter values 10 and 20, which get stored in variable a and b.
- Observe the function call swap(). Instead of passing values of variable a and b, addresses of a and b are passed.
- Since addresses are always stored in pointer variables, we create two pointer variables in function swap() as formal argument.
- Pointer variables x and y received addresses of variable a and b.
- In function swap(), variable a and b of main() function can be accessed with the help of pointers x and y.
- \*x represent variable a and \*y represent variable b.
- Notice that values need not to be returned as changes made in actual locations only.
- Also notice the declaration of function swap(). As two addresses of int block would be passed during function call, it is specified as void swap(int\*, int\*);

### 9.13 Pointers and Arrays

Remember two things about pointers and arrays:

- 1) Whatever is the size of an array it always stored in contiguous memory locations.
- 2) When pointer is incremented it always points to immediately next location of its type.

#### Example:

```
#include<stdio.h>
#include<conio.h>
void input(int *);
void display(int *);
main()
{
 int a[5];
 clrscr();
 input(&a[0]);
 display(&a[0]);
 getch();
}
void input(int *p)
{
 int i;
 printf("Enter five numbers");
 for(i=0; i<=4; i++)
 scanf("%d", p+i);
}
void display(int *p)
{
 int i;
 printf("Five numbers are: ");
 for(i=0; i<=4; i++)
 printf("%d ", *(p+i));
```

```
}
```

## Explanation

- Address of very first block of array that is address of a[0] is passed during function call of input() and display()
- Address is received in pointer variable p.
- We need not to create as many pointer variables as we have array blocks. We need to store address of first block only (a[0]). Rest of the addresses can be obtained by simple pointers arithmetic. By adding 1 to p we get address of a[1], similarly by adding 2 to p, we get address of a[2] and so on.
- In the function input(), see scanf(), we wrote p+i in it. As the value of i increases in every iteration of for loop, p+i represents address of successive blocks of array.
- In function display(), see printf(), we wrote \*(p+i), as we need to pass values of array blocks in function printf().

## Handling two dimensional array via pointers

There are two ways to create pointers that will be used to store address of two dimensional arrays. First is simply create a pointer and second is pointer to an array

The following example illustrates the first way.

```
main()
{
 int i;
 int a[2][3]={3,5,1,6,7,8};
 int *p;
 p=&a[0][0];
 for(i=0;i<=6;i++)
 printf("%d", *(p+i));
}
```

Array of any dimension, always consumes memory in contiguous manner. In our example we made an array of six blocks. Address of the first block gets stored in pointer p. Now by incrementing address contained in p by one each time, we can point to next block in array

Second way is to create a pointer to an array.

```
main()
{
 int i,j;
 int a[2][3]={3,5,1,6,7,8};
 int (*p)[3];
 p=a;
 for(i=0;i<=1;i++)
 {
 for(j=0;j<=2;j++)
 printf("%d ", *(*(p+i)+j));
 }
}
```

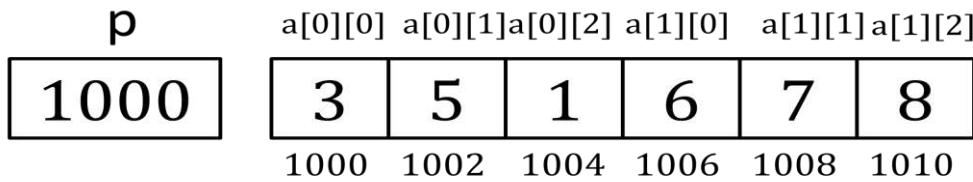
```

 printf("\n");
 }
}

```

Explanation:

- Notice the way we declare pointer. This is not a simple pointer but it is a pointer to an array of 3 int blocks.
- In the line p=a, address of array gets stored in pointer p.



- Consider the above diagram, let us assume the address of array is 1000 and it is stored in pointer p.
- p+1 represent address of second array that is 1006. This happens because p is not a simple pointer to an int, rather it is a pointer to an int array of size 3. Thus every time when pointer is incremented, it points to next array.
- \*(p+1) represent address of first block of first address and if we add 1 to it that is \*(p+1)+1, it means address of next block of the first array.

Chart below gives you a clear picture of pointer jargons and their meaning

| Expression | Meaning | Expression | Meaning |
|------------|---------|------------|---------|
| P          | 1000    | *(p+0)+0   | 3       |
| p+1        | 1006    | *(p+0)+1   | 5       |
| *(p+0)     | 1000    | *(p+0)+2   | 1       |
| *(p+1)     | 1006    | *(p+1)+0   | 6       |
| *(p+0)+1   | 1002    | *(p+1)+1   | 7       |
| *(p+0)+2   | 1004    | *(p+1)+2   | 8       |
| *(p+1)+0   | 1006    | *p         | 1000    |
| *(p+1)+1   | 1008    | **p        | 3       |
| *(p+1)+2   | 1010    |            |         |

## 9.14 Pointers and Strings

Just like an int array, char array can also be managed with pointers. We can store address of char array in char pointer. Now this pointer can be used for any string manipulation

```

main()
{
 char str[]="Welcome";
 char *p;
 p=str;
 printf("%s",p);
}

```

```
}
```

Output:

Welcome

There is a difference between p and str. Pointer p is used to store address of char array. We can change value of p at any instant as it is merely a variable. On the other hand str is not a variable. Name of array like str is always used to represent address of first byte of array. Since str is not a variable operations attempting to change value of str leads to compilation error.

Example: Define a function to calculate length of string

```
int stringlength(char *p)
{
 int i=0;
 while(* (p+i) !='\0')
 i++;
 return(i);
}
```

### 9.15 Array of pointers

Array of pointers is a group of pointers. We create many pointers to point similar lists; we can store addresses of these lists in an array of pointers.

Example

```
main()
{
 int a=3,b=4,c=5,i;
 int *p[3];
 p[0]=&a;
 p[1]=&b;
 p[2]=&c;
 for(i=0;i<=2;i++)
 printf("%d ",*p[i]);
}
```

Output:

3 4 5

In the above example we have three pointers, p[0], p[1] and p[2]. These pointers are used to contain addresses of a, b and c. Thus pointer array means sequence of pointers.

### 9.16 Dangling pointer

A pointer becomes dangling when memory of the block is released whose address is still reside in pointer. It is much like wild pointer. The only difference in wild pointer and dangling pointer is, wild pointers are un-initialized pointers thus containing invalid address. Wild pointer points to unknown location which can be dangerous from programming point of view.

On the other hand dangling pointers are initialized perfectly but due to programmers carelessness it is still pointing to the memory area which was released.

```
main()
{
 int *p;
 {
 int x=4;
 p=&x;
 }
 *p=5;
}
```

See the scope of variable x is limited to the inner block. Pointer p is containing address of variable x. As soon as control comes out from inner block, memory of variable x is released, but the pointer p still contains address of x. Here p becomes dangling

### 9.17 Const pointer and pointer to a const

Const pointer is a pointer whose value cannot be altered.

Pointer to a const is a pointer that stores an address of const variable thus we cannot modify variable data using pointer.

Declaration of pointer to a const:

```
const int x = 4;
const int *p; //or int const *p;
p=&x;
```

The above code tells that the pointer p is a pointer to a const. This means we cannot modify variable x using pointer p.

For example:

```
*p=6; //error: cannot modify const variable
p++; //valid:
```

Another example:

```
int x=4;
const int *p; //or int const *p;
p=&x;

x++; //valid
*p=6; //error: cannot modify const variable
p++; //valid
```

Another example

```
const int x=4;
int *p;
p=&x;
x++; //error: cannot modify const variable
*p=3; //undefined behavior, should be an error
p++; //valid
```

Example for const pointer

```
int x=5;
int *const p=&x;
p++; //error: cannot modify const variable p
x++; //valid
*p=7; //valid
```

Another example:

```
int x=5;
int *const p;
p=&x; //error cannot modify const variable p.
```

Remember that const variables must be initialized during declaration.

### Check Yourself:

#### Objective Questions

- 1) Are the expression `*ptr++` and `++*ptr` same?
  - a. Statement is true
  - b. Statement is false
  - c. True only when ptr is an int pointer
  - d. True only when ptr is pointer to an array
  
- 2) Which of the following statements correct about k used in the below statement?

```
char ****k;
```

  - a. k is a pointer to a pointer to a pointer to a char
  - b. k is a pointer to a pointer to a pointer to a pointer to a char
  - c. k is a pointer to a char pointer
  - d. k is a pointer to a pointer to a char
  
- 3) Find the output of the following program

```
#include<stdio.h>
main()
{
 char str[] = "Hello";
 char *const p = str;
 *p='M';
 printf("%s",str);
}
```

  - a. Mello
  - b. Hello
  - c. Error
  - d. MHello
  
- 4) Find the output of the following program

```
#include<stdio.h>
main()
{
 static char *s[]={ "black", "white", "pink", "violet"};
 char **ptr[]= { s+3, s+2, s+1, s }, ***p;
 p=ptr;
 ++p;
 printf("%s", **p+1);
}
```

  - a. ink
  - b. ack
  - c. ite
  - d. let

- 5) Point out the error in the following code

```
#include<stdio.h>
main ()
{
 int a[] = {10, 20, 30, 40, 50 };
 int j;
 for(j=0; j < 5; j++)
{
 printf("%d ",a);
 a++;
}
}
a. 10 20 30 40 50
c. Error: LValue required
```

- b. 20 30 40 50
d. 50 40 30 20 10

- 6) What would be the equivalent pointer expression for referring the array element  
**a[i][j][k][l]**

- a. (((((a+i)+j)+k)+l)
c. (((a+i)+j)+k+l)

- b. \*(\*(\*(\*a+i)+j)+k)+l)
d. ((a+i)+j+k+l)

- 7) Find the output of the following program

```
#include<stdio.h>
```

```
main()
{
 int a=10,b=20;
 int * (*fp)(int*,int);
 fp=fun;
 printf(" %d %d", *fp(&a,b),a);
}
int* fun(int *p,int q)
{
 *p=*p+q;
 q=*p-q;
 *p=*p-q;
 return(p);
}
```

- a. 10 10
c. 20 20

- b. 20 10
d. 10 20

- 8) Find the output of the following program

```
#include<stdio.h>
```

```
main()
{
 int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
 printf("\n%u %u %u",a[0]+1,*(a[0]+1),*(*(a+0)+1));
} //Assume address of a[0][0] is 1000
a. 1008 1008 1
b. 1008 1008 2
```

- c. 1006 2 2 d. 1006 1006 1

9) Find the output of the following program

```
#include<stdio.h>
main()
{
 char *p="";
 char s[]={};
 printf("%d %d",sizeof(p),sizeof(s));
}
```

a. 2 1 b. 2 2  
c. 1 2 d. 1 1

- 10) Find the output of the following program

```

int * fun(int*,int);
main()
{
 int a=10,b=20;
 printf("%d %d", *fun(&a,b), a);
}
int fun(int *p, int q)
{
 *p=*p+q;
 q= *p-q;
 *p=*p-q;
 return(p);
}
a. 10 20
c. 20 20

```

## **Write following programs**

- 1) Write a program to print address of variable.
  - 2) Write a program to perform input output operations on a variable and should be handled only by pointer.
  - 3) Write a program to swap content of two variables. Variables should be created in main() and should be swapped by function swap(). (hint: Use call by reference)
  - 4) Write a function to calculate area and circumference of a circle but results should be printed in main() function
  - 5) Define functions input(), display() and sort() for the following program:

```
main()
{
 void input(int*);
 void display(int *);
 void sort(int*);
 int arr[10];

 input(arr);
 display(arr);
```

```
sort(arr);
display(arr);
getch();
}
```

The job of function input() is to take data from user for the array. Function display() use to display the content of array. Function sort() is to sort array data in ascending order.

- 6) Write a function to compare two strings and returns 0 if they are same else return ASCII difference. Prototype of function is:  
`int strmatch(char * ,char *);`
- 7) Write a function to reverse a string and returns address of first block of reversed string.
- 8) Write a function to add two matrices of order 3x 3. Use pointer as a formal argument to receive address of an array.
- 9) Write a program to handle three variables of different types with single void pointer
- 10) Write a function to perform case insensitive comparison of two strings.

#### **Explore and Answer the following**

- 1) Write a program to sort a two dimensional array with a pointer to array
- 2) Write a program in which a pointer to array is used to handle a two dimensional array(4x4) containing all zeros except any one block which contains 1. Display values of array in row column fashion. Allow user to move this 1 in the matrix with arrow keys.
- 3) Write a function to convert a string to a number without using predefined function.
- 4) Write a program to print all permutations of a given string

# Chapter 10: Structure and DMA

## Content

- 10.1      Structure
- 10.2      Defining Structure
- 10.3      Using structure
- 10.4      Accessing members of structure
- 10.5      Structure and function
- 10.6      Structure and pointer
- 10.7      DMA
- 10.8      Function malloc()
- 10.9      Function calloc()
- 10.10     Function free()
- 10.11     Link List program
- 10.12     Program to input any length string

### **10.1 Structure**

Structure is a collection of heterogeneous data. Just like an array it is also a group of variables but can be of dissimilar types. Using structure we can create data type known as user defined data type. Such data types are also known as non primitive data types.

Comparing structures with arrays, both are collection of memory blocks used to store multiple data. Array variables do not have names but have indexes, here in structure member variables do have names and can be of different data types.

### **10.2 Defining a structure**

Defining a structure means telling a compiler about this new data type.

### Syntax

```
struct <tag>
{
 Type variableName;
 Type variableName;
 ...
};
```

- struct is a keyword.
- <tag> is any name of your choice. It becomes the name of user defined data type.
- Type is any data type
- Variables declaring inside structure body are called member variables.
- Remember to put semicolon at the end of structure body.

### 10.3 Using structure

Using structure means using new data type defined with the help of struct keyword.

Just like we declare variables of primitive types(int, char, float, etc), we can declare variables of user defined data type.

#### Example

```
#include<stdio.h>
struct student
{
 int rollno;
 char name[20];
};
void main()
{
 struct student s1;
 ...
}
```

- Once you have defined the structure you can use it as a data type in your program.
- Every time you use data type which is defined with the help of struct keyword, must be preceded by struct keyword. Observe the statement in main(), student is prefixed with struct.
- Student is a non primitive or user defined data type
- s1 is a variable of type student.
- S1 occupies 22 bytes in the memory; first 2 bytes for 'rollno' and next 20 bytes are for char array 'name'.
- You can define a structure either inside a function or outside a function. When defined inside a function, it is known as local definition and can only be used by that function only. If the definition resides outside the function, it is known as global

definition and can be used anywhere in the program. In the above example we have use the later one.

- You can also declare variables during defining a structure as follows

```
struct student
{
 int rollno;
 char name[20];
}s1,s2;
```

## 10.4 Accessing members of structure

Accessing structure members is possible via membership operator (.).

Defining a structure doesn't mean creating a variable. No memory will be occupied by the structure until it is used to create variables.

Example

```
#include<stdio.h>
struct student
{
 int rollno;
 char name[20];
};
void main()
{
 struct student s1;
 printf("Enter rollno of student\n");
 scanf("%d",&s1.rollno);
 printf("Enter name of student\n");
 gets(s1.name);
 printf("You have entered:");
 printf("Roll no: %d",s1.rollno);
 printf("Name : %s",s1.name);
}
```

## 10.5 Structure and function

We can pass a structure variable in a function just like we pass values of primitive types.

Similarly we can return a structure.

Following is a sample program to handle student data using functions

Example

```
#include<stdio.h>
struct student
{
```

```

 int rollno;
 char name[20];
 };
 struct student input(void);
 void display(struct student);
 void main()
 {
 struct student s1;
 s1=input();
 printf("You have entered:");
 display(s1);
 }

 struct student input()
 {

 struct student s;
 printf("Enter rollno of student\n");
 scanf("%d",&s.rollno);
 printf("Enter name of student\n");
 gets(s.name);
 return(s);
 }
 void display(struct student s)
 {
 printf("Roll no: %d",s.rollno);
 printf("Name : %s",s.name);
 }
}

```

## 10.6 Structure and pointer

As we know from previous chapters, the type of pointer should be the same as the type of block whose address is contained by that pointer. Thus to contain address of structure variable, the type of the pointer should also be of same type.

```

struct student *p;
struct student s1;
p=&s1;

```

In the above code, p is a pointer of type student, so it can point to any student block. S1 is a variable of type student, hence we can assign address of s1 in p. Accessing members of structure s1 through p need a different operator called member access operator (->), popularly known as arrow operator.

Following is the changed version of input function that takes an address of s1 in structure pointer. It inputs data and store them directly to the variable s1, so no need to return any value.

```

void input(struct student* p)
{
 printf("Enter rollno of student\n");
 scanf("%d", &p->rollno);
 printf("Enter name of student\n");
 gets(p->name);
}

```

### Complete example

```

#include<stdio.h>
struct student
{
 int rollno;
 char name[20];
};
void input(struct student);
void display(struct student);
void main()
{
 struct student s1;
 input(&s1);
 printf("You have entered:");
 display(s1);
}

void input(struct student* p)
{
 printf("Enter rollno of student\n");
 scanf("%d", &p->rollno);
 printf("Enter name of student\n");
 gets(p->name);
}

void display(struct student s)
{
 printf("Roll no: %d", s.rollno);
 printf("Name : %s", s.name);
}

```

## **10.7 DMA (Dynamic Memory Allocation)**

To store data we create variables in our program. This can be done by using data types.

In all our previous programs we wrote data type declaration instruction to specify variables to the compiler. Compiler resolve declaration statements at compile time and maintain variable related information in symbol table.

The amount of memory allocated for variables depend upon the information created by compiler at compile time by looking at declaration statements. This is called static memory allocation.

The benefit of static memory allocation is its cost effectiveness in terms of time. Since decision about amount of memory needed for variables are already done at compile time, no extra time would be consumed for this purpose at the time of execution

The major disadvantage of static memory allocation is lack of flexibility in memory consumption. Since the decision regarding amount of memory has already been taken at compile time no change could be made during run time.

There are scenarios where programmer is not sure about the amount of data to be handled by the program. Static memory allocation is definitely a bad choice in such case. Solution to this problem is dynamic memory allocation.

Decision of allocating memory to store data has taken at run time gives flexibility to the programmer to make program efficient and cost effective in terms of memory consumption

Allocating memory at run time can be done by using either of two predefined functions:

1) **malloc()**

2) **calloc()**

We can also de-allocate memory using a predefined function **free()**. This function can release memory that has been allocated previously by either malloc() or calloc() function

There are subtle differences between static memory allocation and dynamic memory allocation. Memory blocks created by malloc() or calloc() has no name but can only be access using their addresses

### **10.8 Function malloc()**

The malloc() function allocates a block of memory in bytes. The malloc() function is like a request to the RAM of the system to allocate memory, if the request is granted, returns a pointer to the first block. However if it fails to allocate memory returns NULL.

The malloc() function reserves a block of memory of specified size and returns a pointer of type void.

```
ptr=(cast type*)malloc(byte size);
```

### **10.9 Function calloc()**

The `calloc()` function is used for requesting memory space at run time for storing derived data types such as arrays and structures. While `malloc()` allocates a single block of storage space, `calloc()` allocates multiple blocks of storage, each of same size, and then sets all bytes to zero.

```
ptr = (cast type*)calloc(n, element size);
```

This statement allocates contiguous space for n blocks each of size element size bytes. All bytes are initialized to zero and pointer to the first byte of the allocated region is returned. If not enough space NULL is returned

### 10.10 Function free()

The `free` function is used to de-allocate the previously allocated memory using `malloc` or `calloc` functions.

```
free(ptr);
```

### 10.11 Link List Program

**Write a program to prepare a list of integers and manage basic operations on it like adding new data to the list and deleting old data from the list.**

```
#include<conio.h>
#include<stdio.h>
#include<alloc.h>
#include<stdlib.h>
struct node
{
 int x;
 struct node *p;
};
struct node *START=NULL;
void adddata();
void viewlist();
void deletedata();
void main()
{
 int ch;
 while(1)
 {
 clrscr();
 printf("\n1. Add data to list");
 printf("\n2. View list");
 printf("\n3. Delete data from list");
 printf("\n4. Exit");
 printf("\n\nEnter your choice");
 scanf("%d", &ch);
 switch(ch)
 {
```

```

 case 1:
 adddata();
 break;
 case 2:
 viewlist();
 break;
 case 3:
 deletedata();
 break;
 case 4:
 exit(0);
 default:
 printf("Invalid Choice");
 }
 getch();
}
}

void adddata()
{
 struct node *n,*t;
 n=(struct node*)malloc(sizeof(struct node));
 printf("Enter a number: ");
 scanf("%d",&n->x);
 n->p=NULL;
 if(START==NULL)
 START=n;
 else
 {
 t=START;
 while(t->p!=NULL)
 t=t->p;
 t->p=n;
 }
}

void viewlist()
{
 struct node *t;
 if(START==NULL)
 printf("List is empty");
 else
 {
 t=START;
 while(t!=NULL)
 {
 printf(" %d ",t->x);
 t=t->p;
 }
 }
}

void deletedata()
{

```

```

struct node *r;
r=START;
if(START==NULL)
 printf("List is empty");
else
{
 START=r->p;
 free(r);
}
}

```

### **10.12 Program to input string of any length.**

```

main()
{
char ch,*p,*temp=0;
int i=0,size=0;
clrscr();
printf("Enter a string:\n");
p=(char*)calloc(size+1,sizeof(char));
while(1)
{
ch=getch();
if(ch==13)
 break;
if(ch==8)
 continue;

printf("%c",ch);
*(p+i)=ch;
free(temp);
temp=(char*)calloc(size+2,sizeof(char));
for(i=0;i<=size;i++)
{
 (temp+i)=(p+i);
}
*(temp+i)='\0';
free(p);
size++;
p=(char*)calloc(size+1,sizeof(char));
i=0;
while(*(temp+i)!='\0')
{
 (p+i)=(temp+i);
 i++;
}

}
clrscr();

printf("You have entered: %s",temp);

```

```
 printf("\nLength of string is %d", strlen(temp));
 getch();
}
```

## Check Yourself:

### Objective Questions:

- 1) Which of the following are themselves a collection of different data types?
  - a. string
  - b. structures
  - c. char
  - d. All of the mentioned
- 2) Which of the following cannot be a structure member?
  - a. Another structure
  - b. Function
  - c. Array
  - d. None of the mentioned
- 3) Which of the following structure declaration will throw an error?
  - a. struct temp{}s;  
main(){}  
b. struct temp{};  
struct temp s;  
main(){}  
c. struct temp s;  
struct temp{};  
main(){}  
d. None of the mentioned
- 4) What is the output of this C code?

```
#include <stdio.h>
struct student
{
 int no;
 char name[20];
}
void main()
{
 struct student s;
 s.no = 8;
 printf("hello");
}
```

- a. Compile time error
- b. Nothing
- c. hello
- d. Varies

5) #include <stdio.h>  
    struct student  
    {  
int no;  
    char name[20];  
};  
void main()  
{  
    student s;  
    s.no = 8;  
    printf("hello");  
}

- a. Nothing
- b. hello
- c. Compile time error
- d. Varies

6) #include <stdio.h>  
    struct p  
    {  
int k;  
char c;  
float f;  
};  
int main()  
{  
    struct p x = {.c = 97, .f = 3, .k = 1};  
    printf("%f\n", x.f);  
}

- a. Yes
- b. No
- c. Depends on the standard
- d. Depends on the platform

7) Number of bytes in memory taken by the below structure is

```
#include <stdio.h>
struct test
{
 int k;
 char c;
};
```

- a. Multiple of integer size
- b. integer size+character size

- c. Depends on the platform
- d. Multiple of word size

- 8) Which header file should be included to use functions like malloc() and calloc()?
- a. memory.h
  - b. stdlib.h
  - c. string.h
  - d. dos.h
- 9) What function should be used to free the memory allocated by calloc() ?
- a. dealloc();
  - b. malloc(variable\_name, 0)
  - c. free();
  - d. memalloc(variable\_name, 0)
- 10) How will you free the memory allocated by the following program?
- ```
#include<stdio.h>
#include<stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main()
{
    int **p, i, j;
    p = (int **) malloc(MAXROW * sizeof(int *));
    return 0;
}
```
- a. memfree(int p);
 - b. dealloc(p);
 - c. malloc(p, 0);
 - d. free(p);

Write following programs

- 1) Write a program to input some numbers from user and find the maximum value. (Use DMA)
- 2) Write a program to add two complex numbers (a+ib), use structure to define complex number.
- 3) Write a program to manage employee record. Use structure to define an employee. Use functions to manage input and display functionalities.
- 4) Write a memory efficient program to sort n strings in dictionary order

Explore and Answer the following

- 1) Write a program to implement link list. Allow user to choose from following options:
 - (a) add new data, (b) delete particular data, (c) view all data
- 2) Write a program to implement circular queue using link list
- 3) Write a program to implement doubly link list
- 4) Write a program to traverse a tree. (a) In order (b) post order (c) pre order
- 5) Write a program to demonstrate working of stack using link list

Q-1> WHAT IS PYTHON?

ANS:- Python was created by guido van roussum in 1991.
It is general-purpose programming language.
it is interpreted,object-oriented, high-level language.
It is platform independent,widely used in AI,ML.
It is used to develop almost any type of application.
it is easy to learn and require less code to develop the app.
it consists of many libraries and inbuilt functions that make it
easy to maintain and portable.

Q-2> WHAT ARE THE ADVANTAGES OF PYTHON?

ANS:- It is free and open source.
it is portable.
it is extensible.
it is object oriented .
it is built in data structure.

Q-3> WHAT IS PEP 8?

ANS:-PEP 8 is a coding convention which specifies a set of guidelines,
about how to write your python
code more readable.

Q-4> WHAT DO YOU MEAN BY PYTHON LITERALS?

ANS:- Python literals can be defined as data which are given in a
variable or constant.
There are 6 types of literals in python:
String Literals:-
 some text in single or double quote
Character Literals:-
 single character inclosed in quotes
Numeric Literals:-
 integer,float,complex
Boolean Literals:-
 True,False
Special Literals:-
 None
Literal Collections:-
 list,tuple,dictionary,set

Q-5> EXPLAINS PYTHON FUNCTIONS?

ANS:- A function is a section of program or a block of code that is
written once and can be executed
whenever required in the program.
There are two types of functions:
1. Built-in-function
2. User-defined function

Q-6> WHAT IS PYTHON ZIP() FUNCTION?

ANS:- The python zip() function is used to transform multiple lists i.e list1, list2, list3 into a single list of tuples by taking the corresponding elements of lists that are passed as parameters.

Q-7> WHICH ARE THE FILE RELATED LIBRARIES/MODULES IN PYTHON?

ANS:- Python provides libraries/modules with functions that facilitate you to manipulate text files and binary files on file system. By using these libraries, you can create files, update their contents, copy and delete files. These libraries are :- os, os.path & shutil.
OS & OS.PATH:- These libraries include functions for accessing the filesystem.
SHUTIL:- This library is used to copy and delete the files.

Q-8> WHICH ARE THE DIFFERENT FILE PROCESSING MODES SUPPORTED BY PYTHON?

ANS:- python provides three modes to open files:
1.Read-only mode
2.Write-only mode
3.Read-write mode

Q-9> HOW MANY TYPES OF OPERATOR IN PYTHON?

ANS:- There are six types of operator in python.
1.Arithmetic operator
2.Relations operator
3.Assignment operator
4.Logical operator
5.Membership operator
6.Identity operator
7.Bitwise operator

Q-10> WHAT IS OPERATOR IN PYTHON?

ANS:- An Operator is a particular symbol which is used on some values and produces an output as result
for ex:- $3+4=7$
here + and = are operators.

Q-11> HOW IS PYTHON INTERPRETED LANGUAGE?

ANS:- python is an interpreted language, which means the source code of a python program is converted

into bytecode that is then executed by the python virtual machine.
Program written in interpreted language does not need any compilation step.

Q-11> WHAT ARE DIFFERENCES BETWEEN LISTS AND TUPLES IN PYTHON?

ANS:- Lists and Tuples both are sequential data types that can store a collection of data in python.

The data stored in both sequences can have different data types.

Lists are represented with square brackets while tuples are represented with parenthesis.

The key difference between the two is that lists are mutable data structure,

Tuple on the other hand are immutable. This means that lists can be modified, but

Tuple remain constant and cannot be modified in any manner.

Q-12> WHAT IS __INIT__ IN PYTHON?

ANS:- __INIT__ is a reserved method in python classes. It is known as constructor in python classes.

Constructor is used to create instant member. This method automatically called when an

object of that class is created.

In java and c++ constructor name is same as the class name but in python it is compulsory to

create constructor with the name __init__.

Q-13> WHAT IS THE USE OF BREAK AND CONTINUE ?

ANS:- These two are keywords. Break and Continue statements can alter the flow of a normal loop.

BREAK:-

The break statement terminates the loop. Control of the program flow outside the body of the loop.

CONTINUE:-

The continue statement is used to skip the rest of the code inside a loop for the current iteration

only. Loop does not terminate but continues on with the next iteration.

Q-14> WHAT IS SLICING IN PYTHON?

ANS:- As the name suggests, 'slicing' is taking part of.

Slicing in python is a feature that enables us to access a part of sequences like:-

string, tuples, and lists.

syntax of slicing:-

inside subscript operator we mention starting index then colon then end index then colon

then step. step is optional by default its value is +1.

Q-15> WHAT IS THE DIFFERENCE BETWEEN PYTHON ARRAY AND LIST ?

ANS:- Array in python can only contains elements of same data type i.e., data type of array should be homogenous.
List in python can contain elements of different data types i.e., data type of list can be heterogenous.
Arrays are fast while lists are slow.

Q-16> HOW ARRAY IS CREATED IN PYTHON?

ANS:- Array in python can be created by importing array module.
inside array module we have to call array function and pass data_type and value list to create an array with that data type and with the given list value
syntax:- import array as arr
a=arr.array('i',[10,20,30])

Q-17> HOW TO CREATE CONSTANT VARIABLE IN PYTHON?

ANS:- First we have to install pconst library from pip.
then we have to import const module from pconst.
with the help to const module we can create constant variable in python.
for Ex:-if we want to create constant variable p,
then const.p=10
Here p is the constant variable so we cannot change the value of p.

Q-18> WHAT IS LAMBDA IN PYTHON?

ANS:-Lambda is an anonymous function in python that can accept any number of arguments but can only have a single expression.
SYNTAX:-
mul=lambda x,y:x*y
TO CALL:-
mul(4,5)

Q-19> WHAT IS TYPECASTING IN PYTHON?

ANS:- Typecasting means changing one data type into another is known as typecasting.

Q-20> WHAT ARE TUPLE IN PYTHON?

ANS:- Tuple is a collection data type and it is unchangable, it means that we cannot change the value of tuple.

Q-21> WHY PYTHON IS USED FOR FAST DEVELOPMENT CYCLE?

ANS:- python is different from major compiled languages, such as C and C++.

As the code is not needed to be compiled, python code can be readily changed and executed. This helps for a fast development cycle.

Q-22> WHY PYTHON IS SLOWER THAN C AND C++?

ANS:- Since the code is not directly compiled and executed and an additional layer of the python virtual machine is responsible for execution, Python code runs a little slower than conventional language like C, C++, etc.

Q-23> WHAT IS A DYNAMICALLY TYPED LANGUAGE?

ANS:- Dynamic means data types are checked during execution. Python is an interpreted language.

It executes each statement line by line and thus type checking is done on the fly, during execution. Hence, Python is a Dynamically Typed Language.

Q-24> WHAT IS STATICALLY TYPED AND DYNAMICALLY TYPED LANGUAGE?

ANS:- Statically-typed languages perform type checking at compile-time, while dynamically-typed languages perform type checking at run-time.

Statically-typed languages require you to declare the data types of your variables before you use them, while dynamically-typed languages do not.

Q-25> WHAT ARE THE TYPES OF SCOPE VARIABLE?

ANS:- There are commonly two types of scope variables:-

LOCAL SCOPE:-

The variables which are defined in the function are known as local variables.

These variables are defined in the function body and can only be accessed inside

the function body, where it is declared.

GLOBAL SCOPE:-

The variable which can be read or accessed from anywhere in the program is known as global scope.

These variables can be accessed inside and outside the function.

Q-26> WHAT ARE THE BENEFITS OF USING PYTHON?

ANS:- The benefits of using python are:-

- 1.Easy to use
 - 2.interpreted language
 - 3.Dynamically typed language
 - 4.Free and Open source
 - 5.Extensive support for libraries
 - 6.Portable
-

Q-27> WHAT ARE DICTIONARY AND LIST COMPREHENSIONS?

ANS:- Dictionary and list comprehensions are just another concise way to define dictionary and list.

EXAMPLE:-

```
X=[i for i in <collection>]
```

Q-28> WHAT ARE THE COMMON BUILT-IN DATA TYPES IN PYTHON?

ANS:-The common built in data types in python are:-

- 1.NUMBER
 integer,float,complex
 - 2.LIST
 - 3.TUPLE
 - 4.STRING
 - 5.SET
 - 6.DICTIONARY
 - 7.BOOLEAN
-

Q-29> WHAT IS THE DIFFERENCE BETWEEN .PY AND .PYC FILES?

ANS:- The .py files are the python source code files.while the pyc files contain the bytecode of the python files.The interpreter converts the source.py file to pyc.files.

Q-30> WHAT IS NAMESPACE IN PYTHON?

ANS:- A namespace is a naming system used to make sure that names are unique to avoid nameing conflicts.

Q-31> HOW IS MEMORY MANAGED IN PYTHON?

ANS:-Memory management in python is managed by PYTHON PRIVATE HEAP SPACE.The programmer does not have access to this private heap. The python interpreter takes care of this. The allocation of heap space for python objects is done by Python's memory manager. Python also has an INBUILT GARBAGE COLLECTOR,which recycles all the unused memory and so that it can be available to the heap space.

Q-32> WHAT ARE THE POSITIVE AND NEGATIVE INDICES?

ANS:-In the positive indices are applied the search begins from left to the right.In the case of negative indices,the search begins from right to left.
for ex:-
Let the array list of size "n".In case of positive index the first index is 0,then comes 1 and until the last index is -1.
However in case of negative index the first index is -n,then -(n-1) until the last index will be -1.

Q-33> WHAT ARE PYTHON MODULES? NAME SOME COMMON BUILT-IN-MODULE IN PYTHON?

ANS:-Python modules are files containing python code.This code can either be funtions classes or variables.
A Python module is a .py file containing executable code.
Some of the commonly used built in modules are:-
1. OS
2.SYS
3.MATH
4.RANDOM

Q-34> WHAT ARE LOCAL VARIABLES AND GLOBAL VARIABLES IN PYTHON?

ANS:-GLOBAL VARIABLES:-
Variables declared outside a function or in global space are called global variable.
These variables can be accessed by any function in the program.

LOCAL VARIABLES:-
Any variable declared inside a function is known as a local variable. This varible is present in the local space and not in the global space.
When you try to access the local varible outside the any function,it will throw an error.

Q-35> IS INDENTATION REQUIRED IN PYTHON?

ANS:- Indentation is necessary for python. It specifies a block of code. All code within loops, classes, functions is specified within an indented block. It is usually done using four space character. If your code is not indented necessarily, it will not execute accurately and will throw errors as well.

Q-36> HOW DO YOU MODIFY A STRING IN PYTHON?

ANS:- You cannot modify a string because it is an immutable data type. But if we want to modify then you have to first convert it into a list, then update it by subscript operator and then by using join method of string. We can again convert it into a string. AS:-

```
s = list("Hello world")
print(s)
['H', 'e', 'l', 'l', 'o', ' ', 'z', 'o', 'r', 'l', 'd']

s[6] = 'W'

print(s)
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']

m= "".join(s)
print(m)
Hello world
```

Q-37> WHAT IS THE DIFFERENCE BETWEEN LIST METHOD APPEND() AND EXTEND()?

ANS:-
APPEND:- Adds an element to a list.
EXTEND:- Concatenates the first list with another iterable (not necessarily a list, it can be any iterable).

```
x = [1, 2, 3]
x.append([4, 5])
print(x)
[1, 2, 3, [4, 5]]

x = [1, 2, 3]
x.extend([4, 5])
print(x)
[1, 2, 3, 4, 5]

x=[1,2,3]
x.extend((4,5))
print(x)
[1,2,3,4,5]
```

Q-38> WHY WE USE THE PASS STATEMENT IN PYTON?

ANS:- The pass statement is used as a placeholder for future code. When the pass statement is executed,

nothing happens, Python has the syntactical requirement that code blocks cannot be empty.
Otherwise, it will occur syntax error.

Q-39> WHAT IS CALLABLE?

ANS:- A callable is anything that can be called.
A callable object allows you to use round parenthesis () and eventually pass some parameters,
just like functions.

Q-40> EXPLAIN SPLIT() AND JOIN() FUNCTION?

ANS:- You can use split() function to split a string to a list of strings based on a delimiter.
You can use join() function to join a list of strings to give a single string based on a delimiter.

Q-41> WHAT IS PANDAS IN PYTHON?

ANS:- Pandas is an open source python package that is mostly used for data science/data analysis and machine learning. It has a very rich set of data structure for data based operations.
It provides two fantastic data structure:- series and dataframes.
DATAFRAMES:- it is mutable datatypes present in pandas library.
It provides support for heterogenous data.
PANDAS SERIES:- series is one dimensional pandas data structure present in pandas library that supports data of almost any types.
It supports multiple operations and is used for single dimensional data operation.

Q-42> HOW DO YOU ADD SINGLE-LINE AND MULTI-LINE COMMENTS IN PYTHON?

ANS:- SINGLE LINE COMMENTS:- # is used to denote single line comments.
MULTI-LINE COMMENTS:- """ """(Triple quotes are used across many lines)

Q-43> WHAT IS THE EFFICIENT WAY TO ADD ELEMENT TO A TUPLE OR STRING?

ANS:- Tuples are immutable datatypes in python.
* you cannot add elements to an existing tuple.
* A new one must be created if the value are to be changed!

Q-44> WHAT IS DICTIONARY IN PYTHON?

ANS:- A Python dictionary is a collection of items with keys and values pair. It is written in curly brackets.

They are optimized to retrieve value for known keys.

Q-45> WHAT IS WITH STATEMENT USED FOR IN PYTHON?

ANS:-

Q-46> WHAT IS A MAP() FUNCTION IN PYTHON?

ANS:- The map() function in python is used for applying a function on all elements of a specified iterable.

* it consists of two parameters, function_name and iterable.

Q-47> WHAT IS DIFFERENCE BETWEEN NUMPY AND SCIPY?

ANS:- Numpy stands for Numerical python while Scipy stands for Scientific python.

* Numpy is the basic library for defining array and simple mathematical problems.

* Scipy is used for more complex problems like numerical integration and machine learning and so on.

Q-48> HOW TO REMOVE SPACES FROM A STRING IN PYTHON?

ANS:- Spaces can be removed from a string in python by using strip() or replace() functions.

* Strip() function is used to remove the leading and trailing white space while the replace() function
is used to remove all the white spaces in the string.

```
for ex:- a="ravi kumar"
         b=a.replace(" ", "");
         print(b) --> ravikumar
```

Q-49> EXPLAIN THE FILE PROCESSING MODES THAT PYTHON SUPPORTS?

ANS:- There are six file processing modes in python:

1. read-only mode(r)
 2. write-only mode(w)
 3. append-only mode(a)
 4. r++ mode
 5. w++ mode
 6. a++ mode
-

Q-50> EXPLAIN ALL THE MODES?

ANS:-

Q-51>WHAT IS UNNITEST IN PYTHON?

ANS:- Unittest is a unit testing framework in python.

Q-52> HOW DO YOU DELETE A FILE IN PYTHON?

ANS:- We can delete a file by importing OS module.
ex:- `os.remove(filename)`

Q-53> HOW DO YOU CREATE AN EMPTY CLASS IN PYTHON?

ANS:- To create an empty class we can use the pass command after the definition of the class object.

A pass is a statement in python that does nothing.

Q-54> WHAT ARE PYTHON DECORATORS?

ANS:- Decorators are functions that take another functions as argument to modify its behaviour without changing the function itself.

* These are useful when we want to dynamically increase the functionality of a function without changing it.

Q-55> WHAT DOES EARLY BINDING MEANS?

ANS:- Early binding is a popular concept in OOPS that is used for many purposes.

* Majorly, it is used to assign values during compile time to save a lot of time during the execution of the entire project.
