

# 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 ?

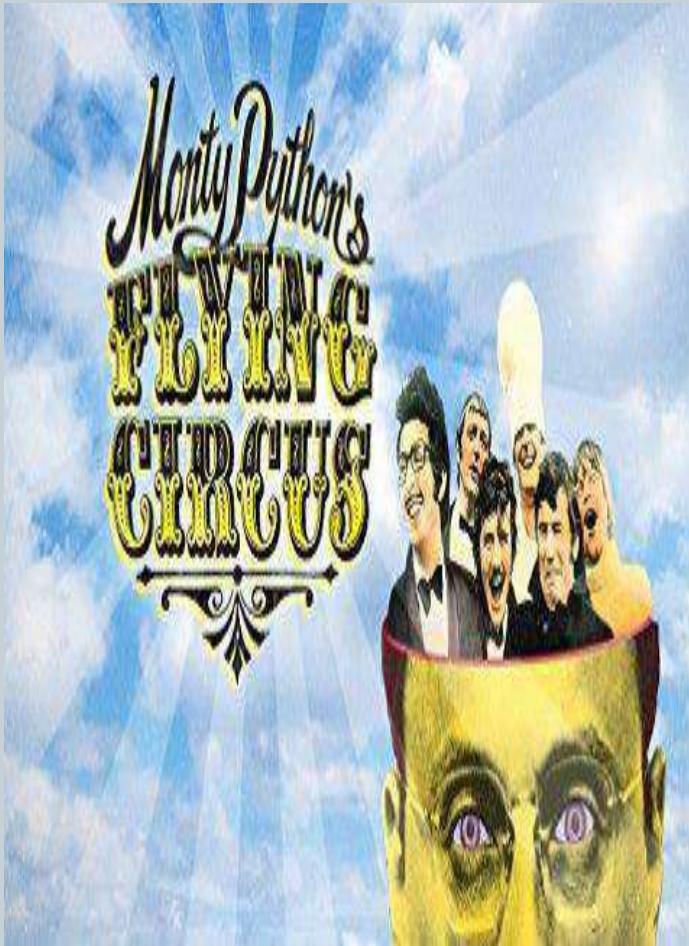


**Guido** started **Python** development as a **hobby** in **1989**

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**

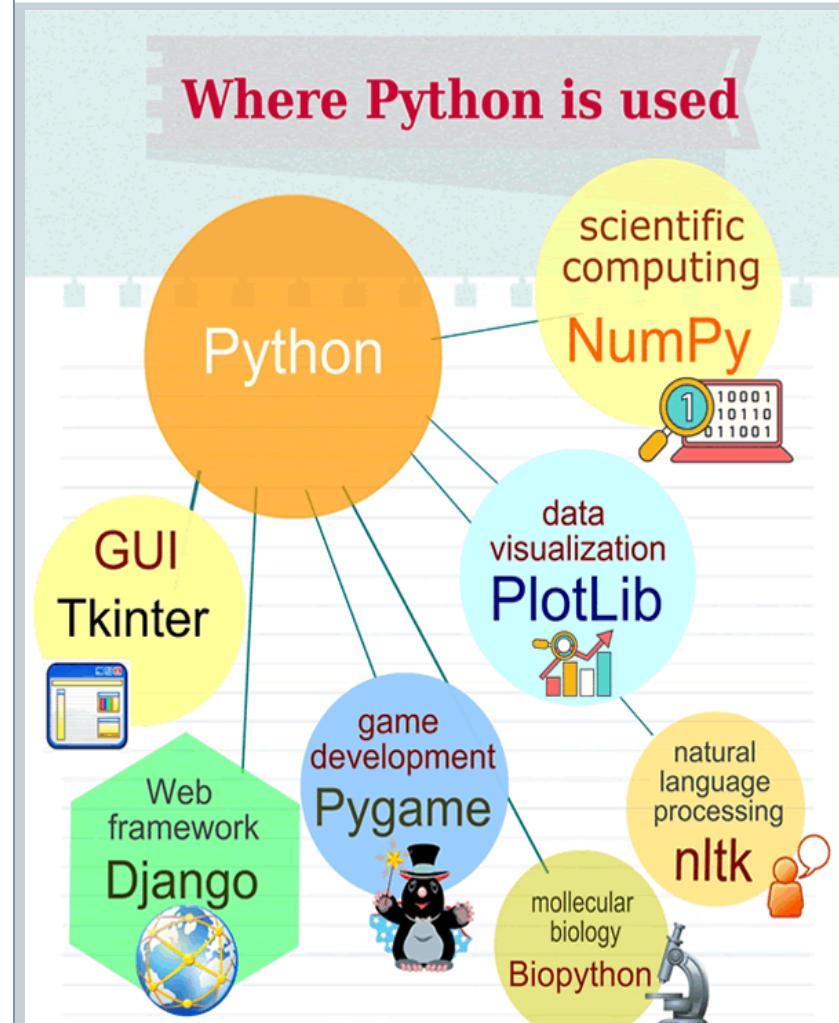


It is **a non-profit organization**  
devoted to the growth and  
enhancement of **Python**  
language

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



# Where Is Python used?



GUI

Web

Data Science

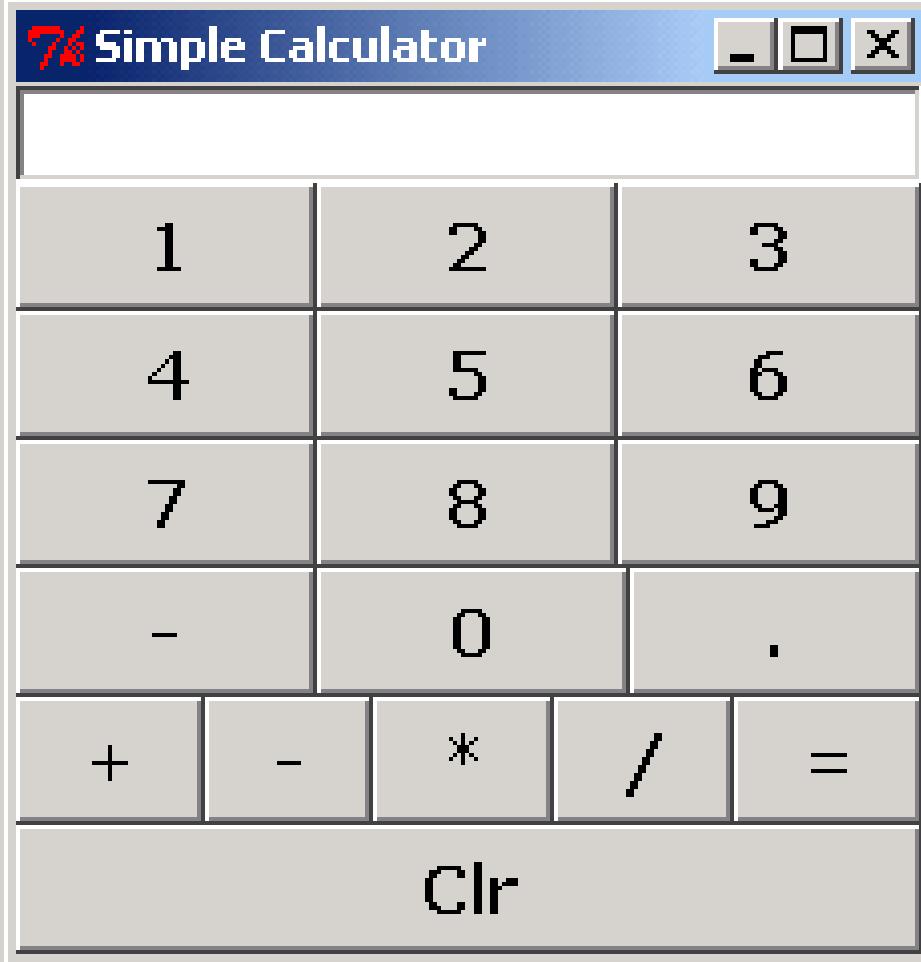
AI & ML

IoT

Hacking



# GUI In Python

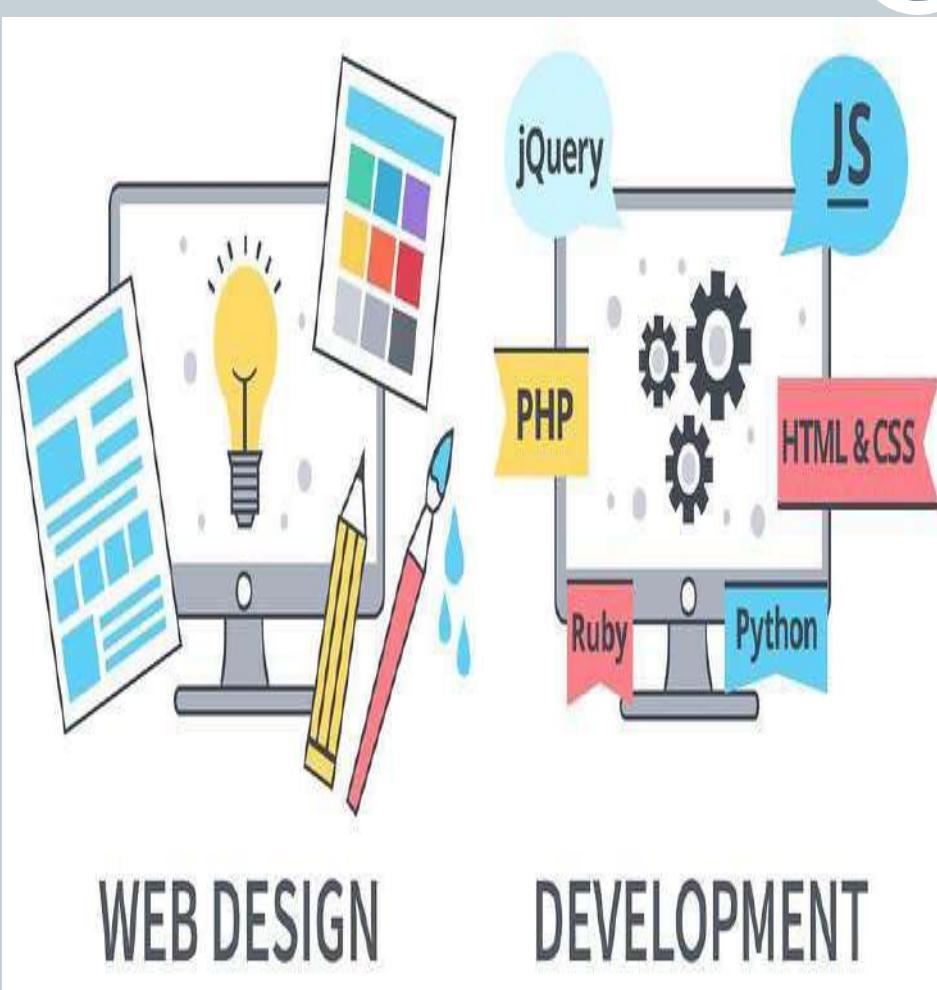


Python is used for GUI apps all the time.

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.



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



Reviewed by Emily Henderson, B.Sc.

May 20 2020



6



6



6



6

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.

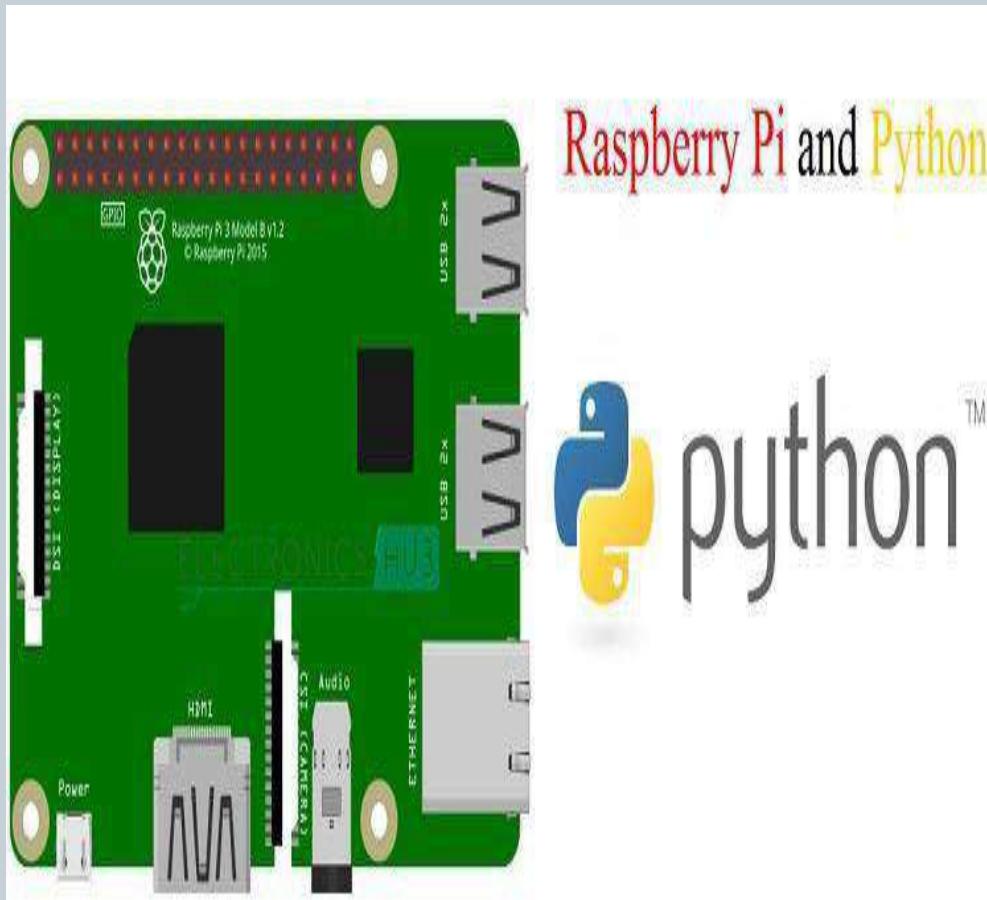
## For example:

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



# IoT In Python

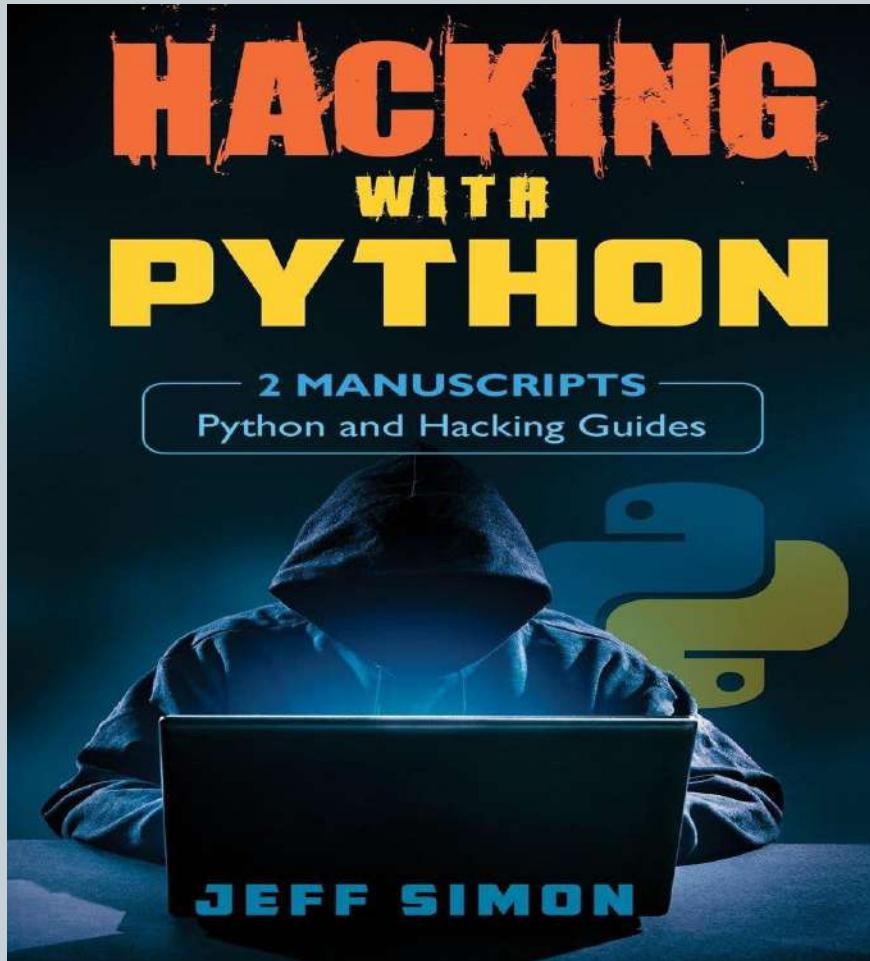
We can build **Home Automation System** and even **robots** using **IoT**



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



# Hacking In Python

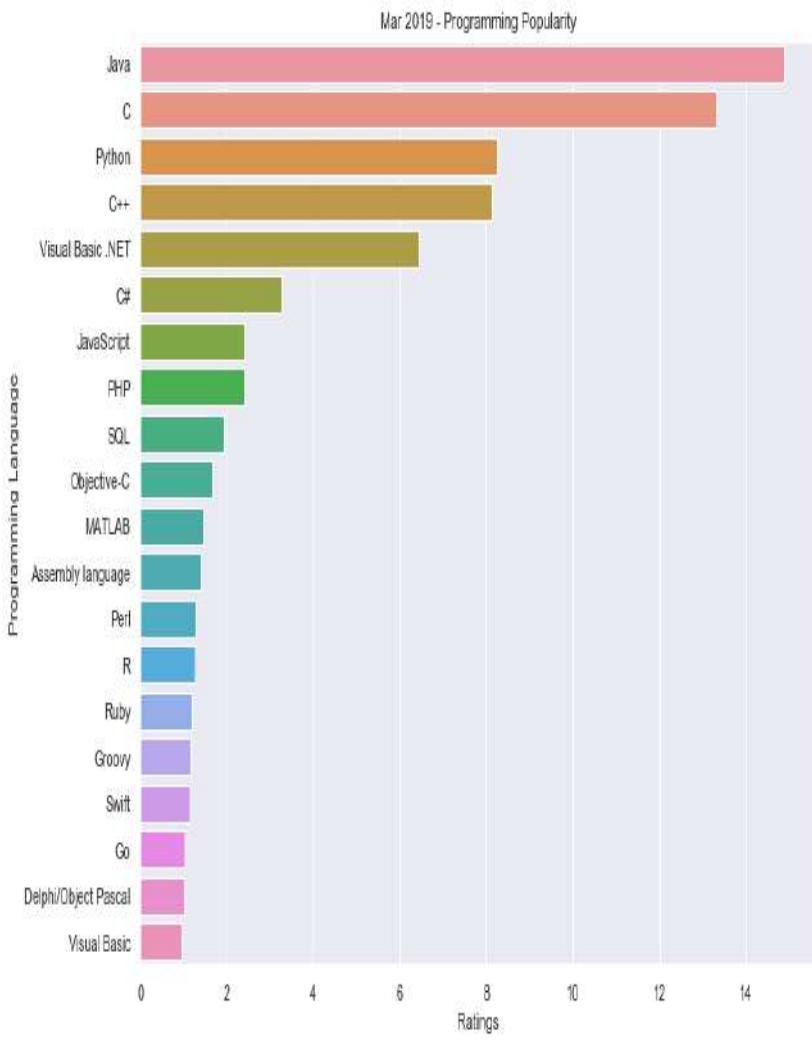


Python has gained popularity as preferred language for hacking.

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



# Why should I learn Python ?



3<sup>rd</sup> 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

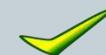
IN C

```
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-20<sup>th</sup> -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-28<sup>th</sup>-2018
- Python 3.7 launched on June-27<sup>th</sup> -2018
- Python 3.8.0 launched on Oct-14<sup>th</sup> -2019
- Current latest (as of now) is Python 3.8.5 launched on Jul-20<sup>th</sup>-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

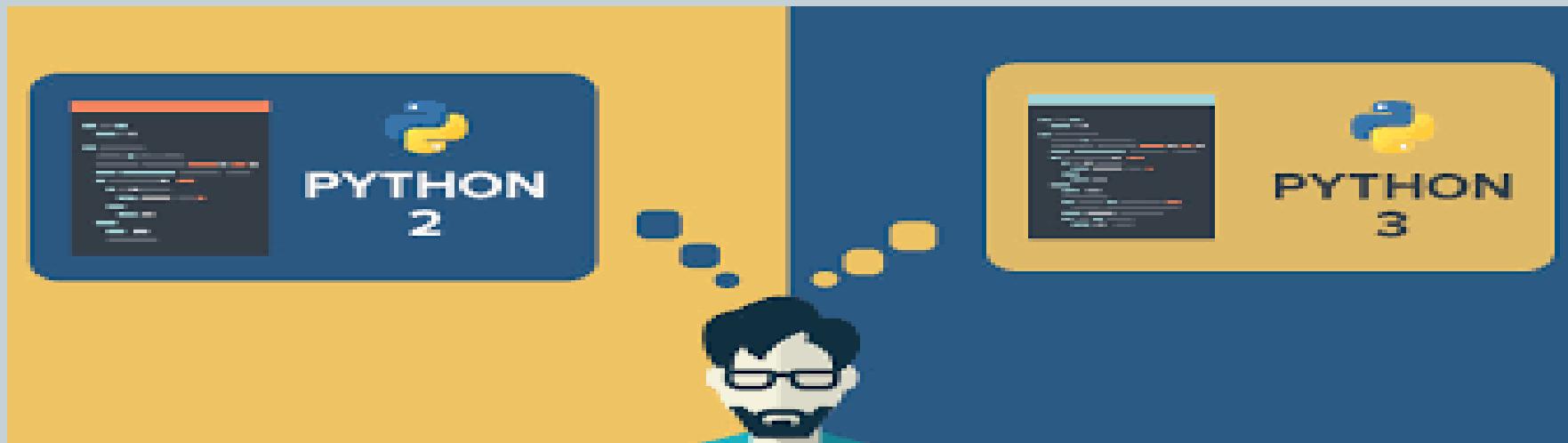


- So to prevent this overhead of programmers , **PSF** decided to support **Python 2** also.
- But this support was only till **Jan-1-2020**
- You can visit **<https://pythonclock.org/>** to see exactly how much time before **Python 2** has retired.

# 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**



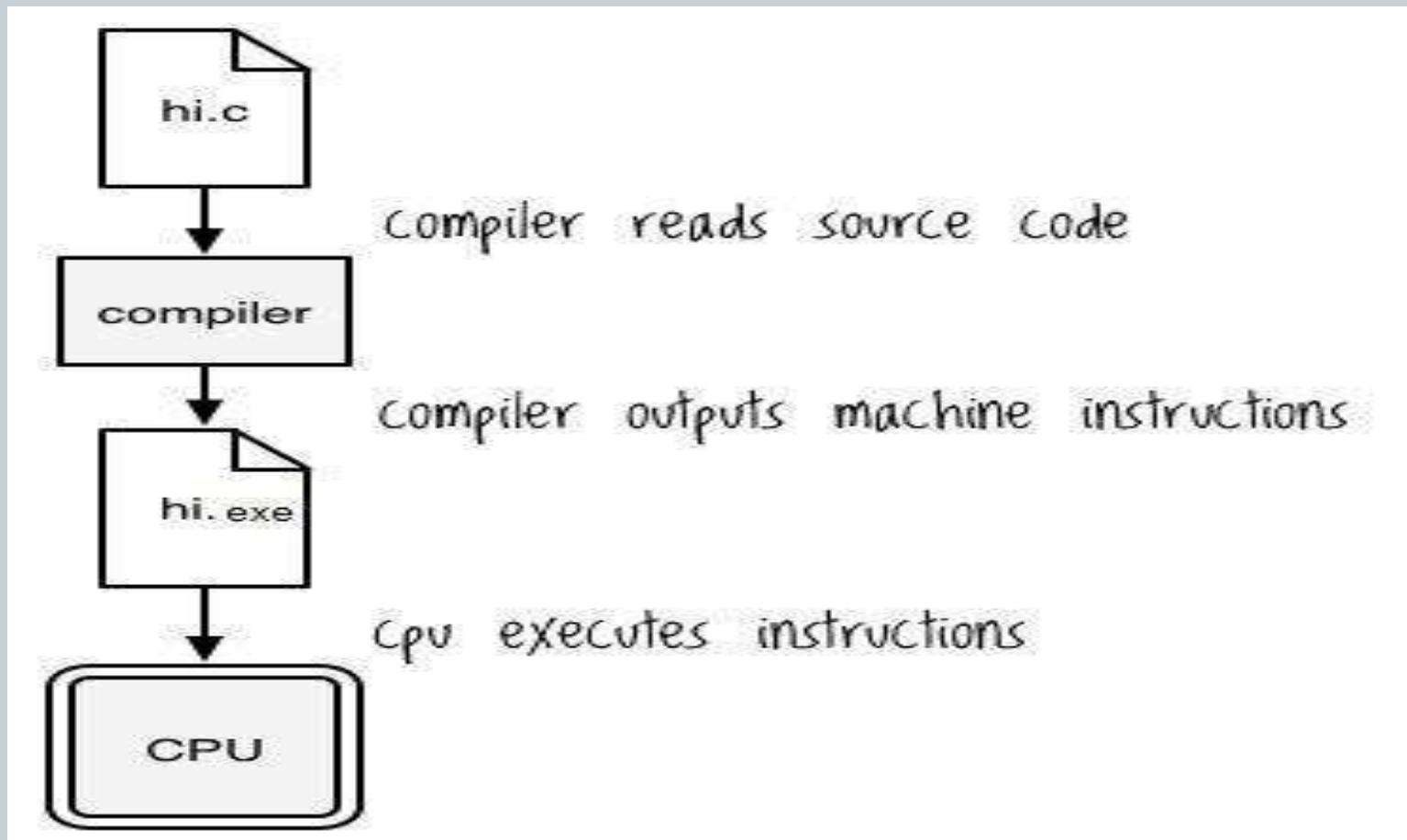
# Various Implementations Of Python

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

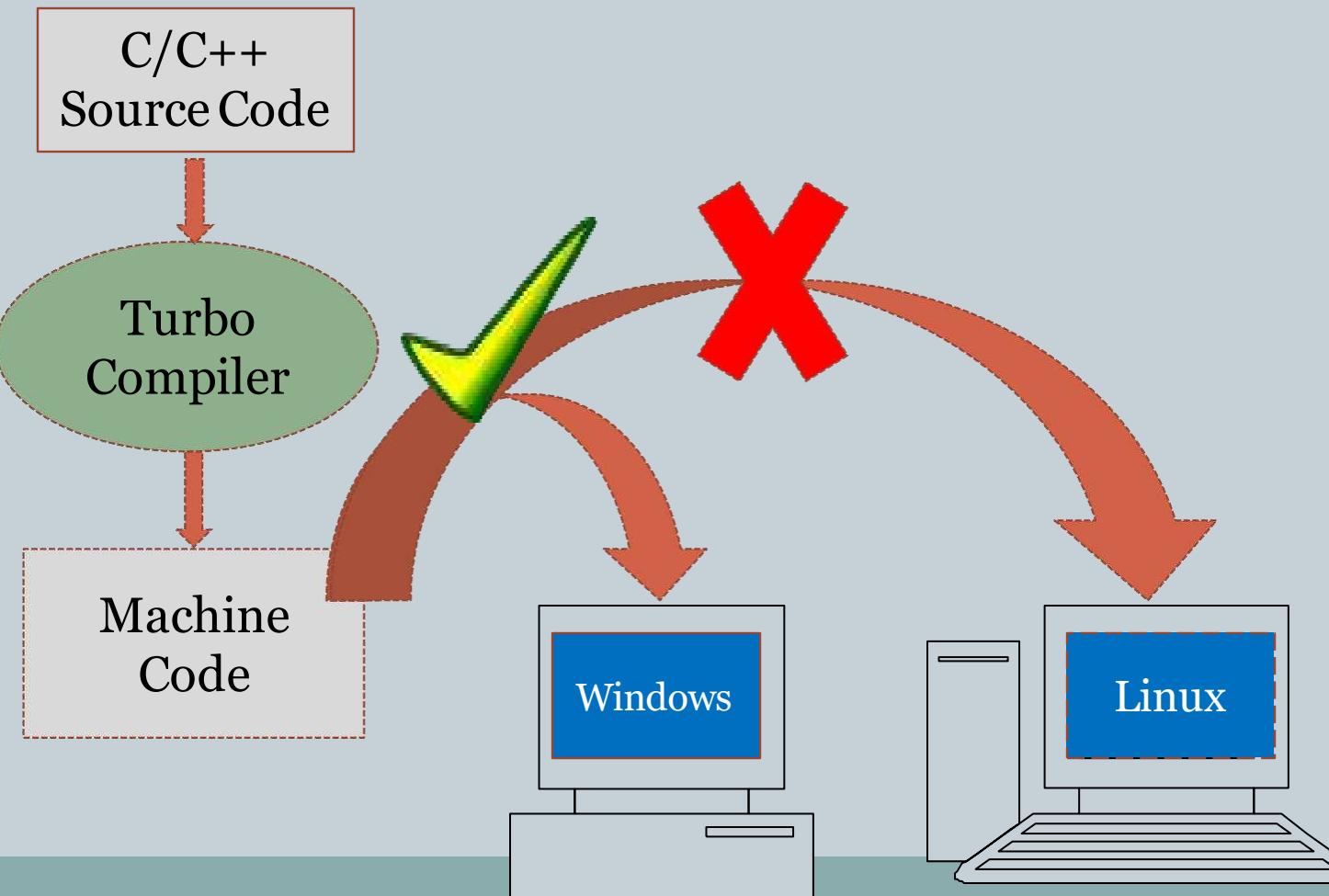


The benefit of **machine code** is that **it runs very fast** because it is in the **native form** i.e a form which is directly understandable to the **CPU**.

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**.

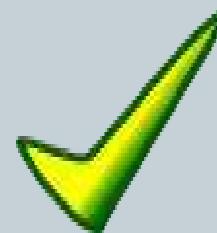
# Program Execution in Python



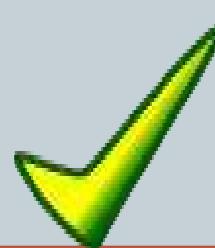
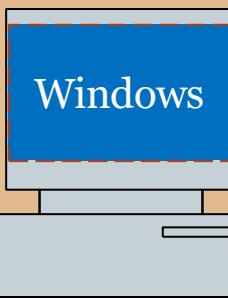
Source Code



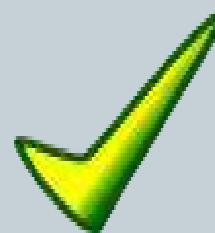
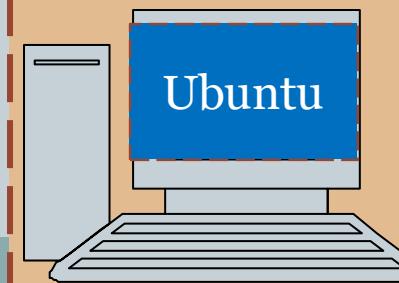
Byte Code



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<sub>P</sub>ython



The **default implementation** of the **Python** programming language is **C<sub>P</sub>ython** which is written in **Clanguage**.

**C<sub>P</sub>ython** is the original **Python** implementation and it is the implementation we will download from [\*\*Python.org\*\*](https://www.python.org).

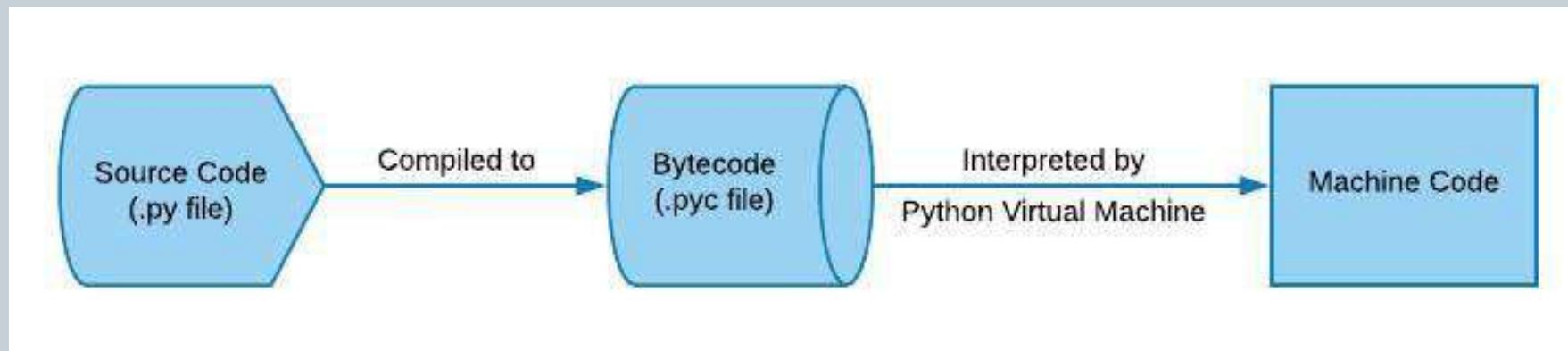
People call it **C<sub>P</sub>ython** to distinguish it from other **Python** implementations

Also we must understand that **Python** is the language and **C<sub>P</sub>ython** is its **compiler/interpreter** written in **Clanguage** to run the **Python** code.



# C<sub>P</sub>ython

**C<sub>P</sub>ython** compiles the **python source code** into intermediate **bytecode**, which is executed by the **C<sub>P</sub>ython virtual machine** also called as the **PVM**.





# Jython



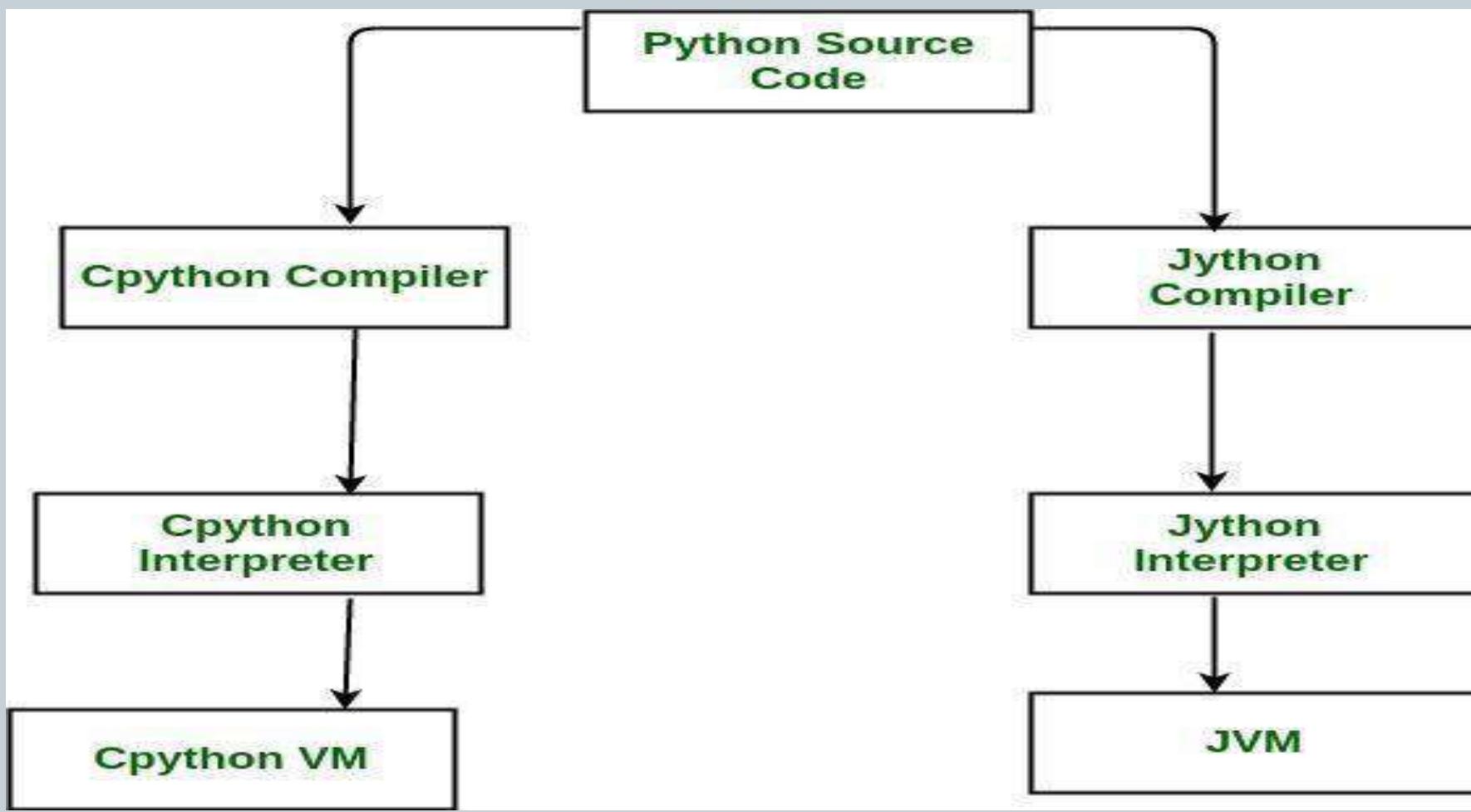
The **Jython** system (originally known as **JPython**) is an alternative implementation of the **Python** language, targeted for integration with the **Java** programming language.

**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.

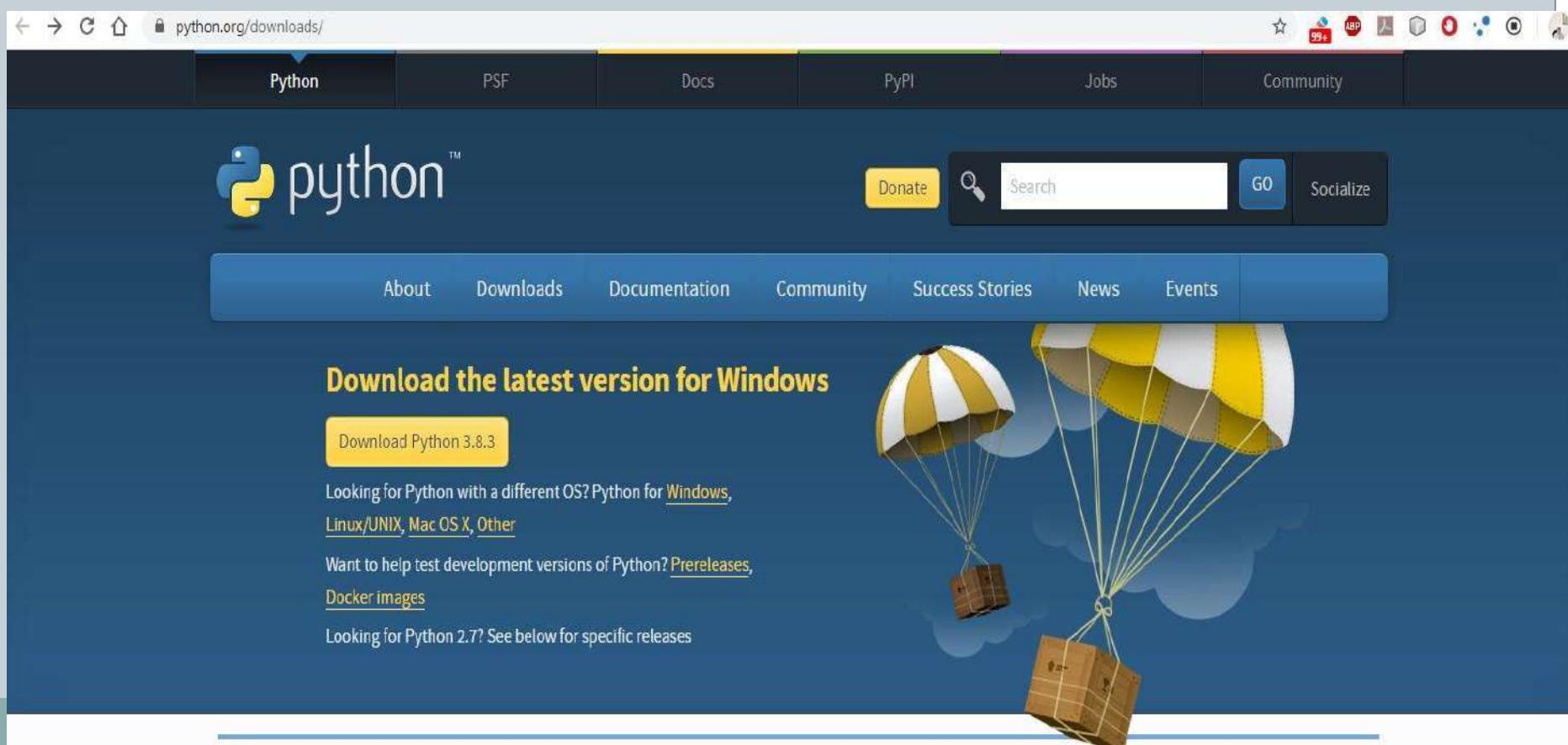
We can download everything we need to get started with **Python** from the **Python** website called <http://www.python.org/downloads>

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 header features the Python logo and navigation links for Python, PSF, Docs, PyPI, Jobs, and Community. A search bar with a magnifying glass icon and a 'GO' button is also present. The main content area has a dark blue background with white text. It features a large yellow button labeled 'Download Python 3.8.3'. Below this, there are sections for 'Downloads' (with links to Windows, Linux/UNIX, Mac OS X, and Other), 'Prereleases' (with Docker images), and specific releases for Python 2.7. To the right, there's a cartoon illustration of two boxes descending from the sky on parachutes. At the bottom, there's a link to 'Active Python Releases'.

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

Active Python 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	<a href="#">Download</a> Release Notes
Python 3.8.3rc1	April 29, 2020	<a href="#">Download</a> Release Notes
Python 2.7.18	April 20, 2020	<a href="#">Download</a> Release Notes
Python 3.7.7	March 10, 2020	<a href="#">Download</a> Release Notes
Python 3.8.2	Feb. 24, 2020	<a href="#">Download</a> Release Notes
Python 3.8.1	Dec. 18, 2019	<a href="#">Download</a> Release Notes
Python 3.7.6	Dec. 18, 2019	<a href="#">Download</a> Release Notes
Python 3.6.10	Dec. 18, 2019	<a href="#">Download</a> 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

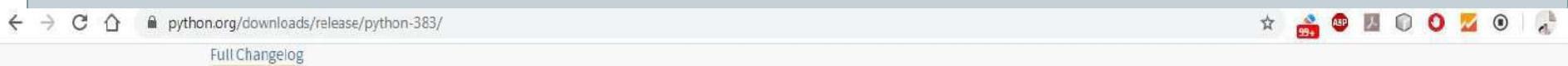
Python was created in the early  
1990s by Guido van Rossum at  
Stichting Mathematisch Centrum in

# 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 highlighted "Python 3.8.2" text. The command prompt shows the path "C:\Users\Sachin>" at the bottom.

```
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?

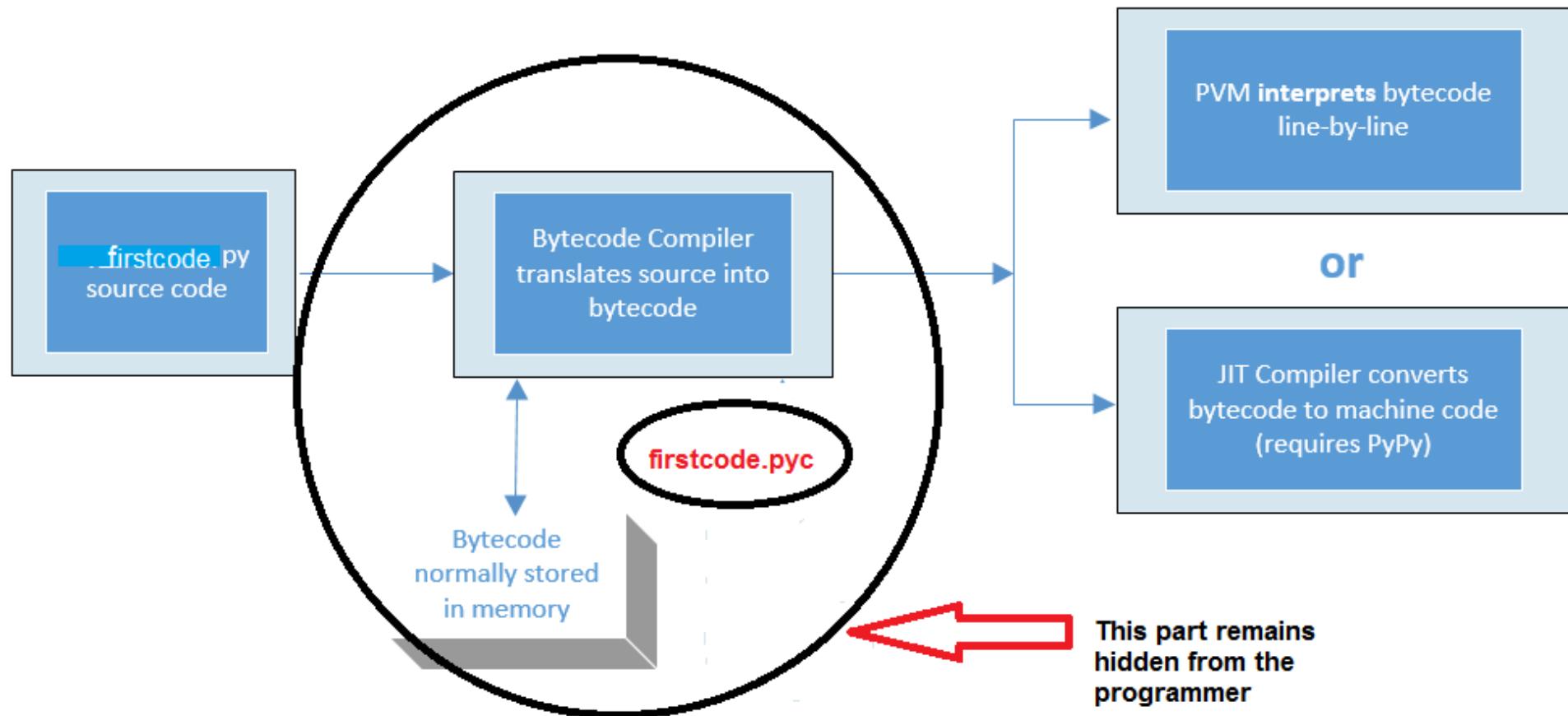


- So in our case the following sequence of events will take place:
  1. Programmer writes **the source code** by the name **firstcode.py**
  2. Then he/she **runs the program** using the command:  
**python firstcode.py**
  3. The Python compiler internally generates **the bytecode file** called as **firstcode.pyc**
  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")
```

A screenshot of a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window shows the command 'D:\My Python Codes>python firstcode.py' being run, followed by the output 'Hello UserPython Rocks'. The prompt then changes to 'D:\My Python Codes>'.

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

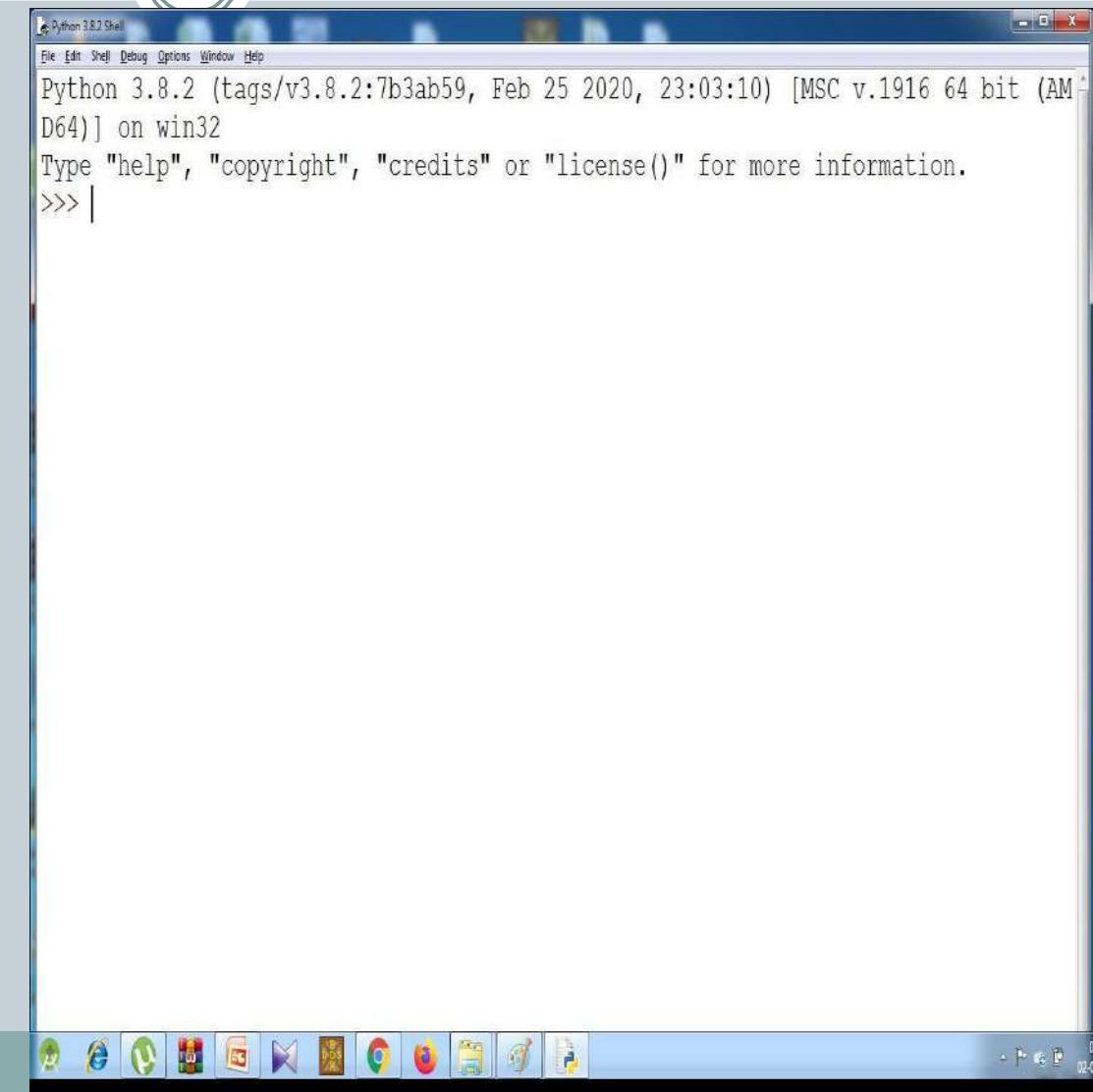
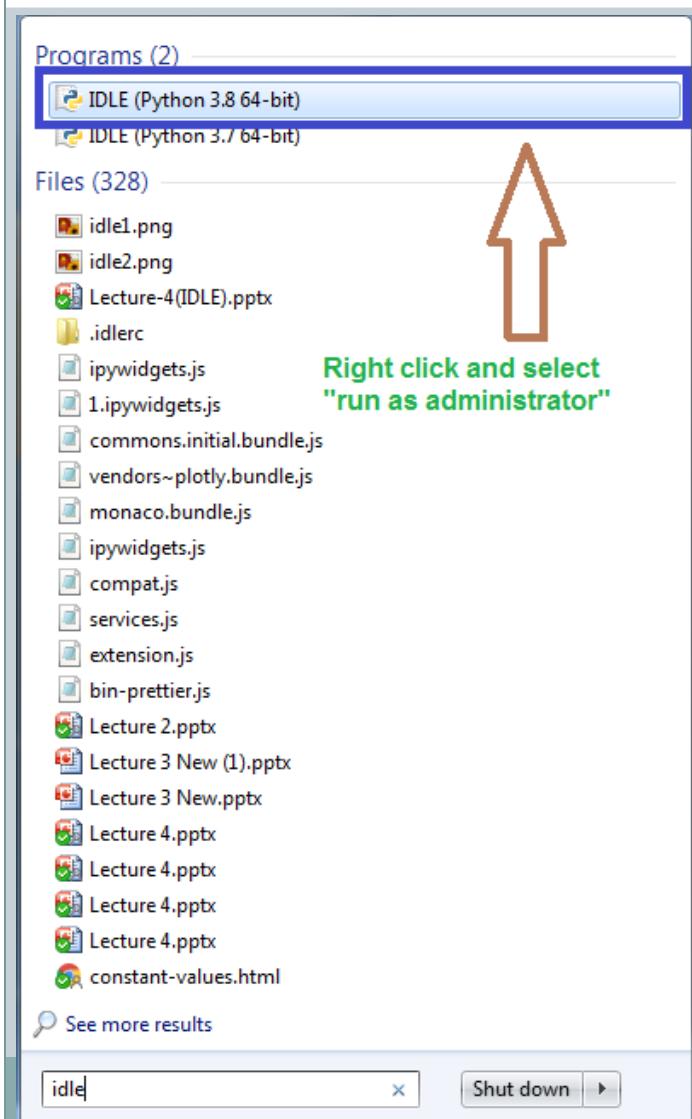


To start **IDLE** on Windows click the **Start Menu** and search "**IDLE**" or "**idle**".

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

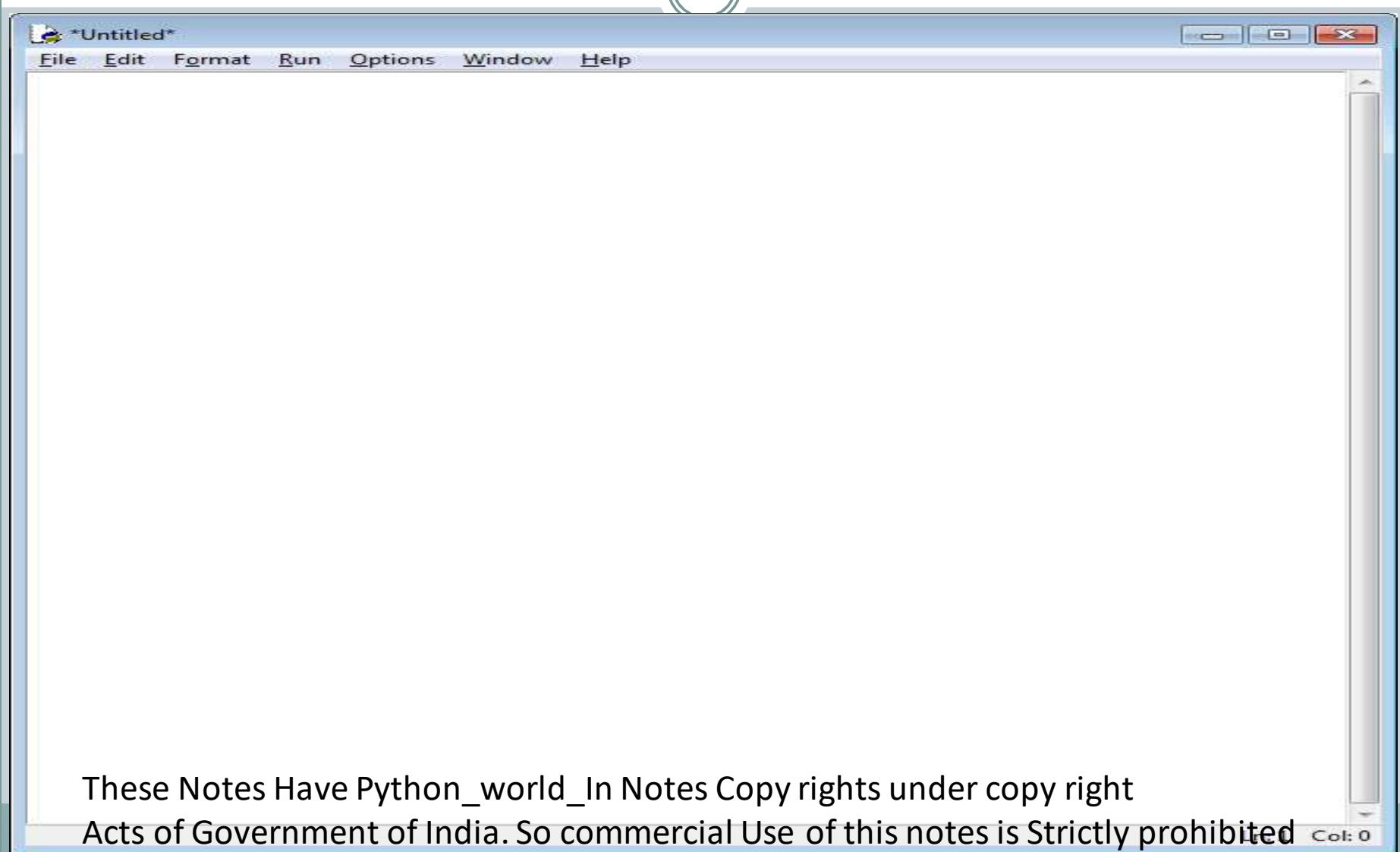
IDLE also has a **built-in text editor** to write Python programs.

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



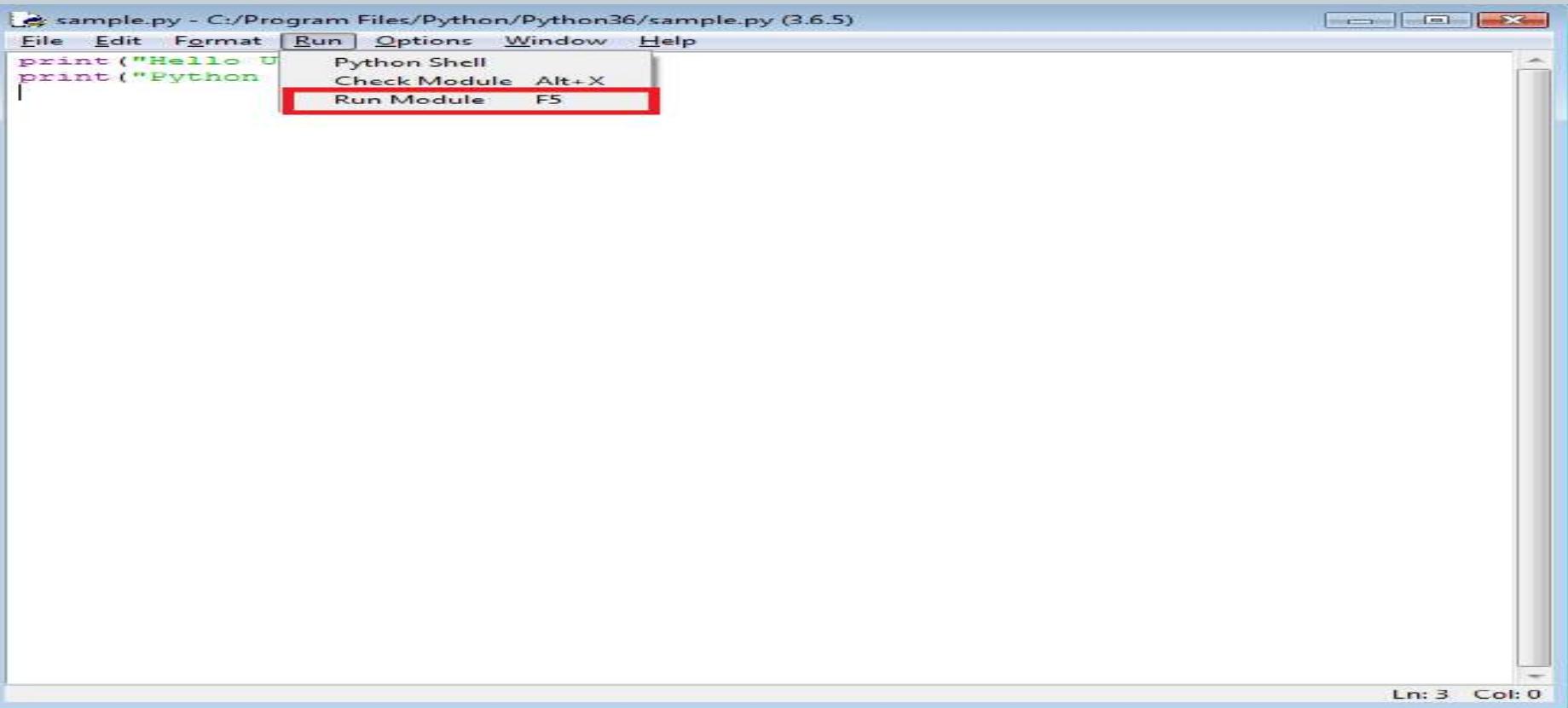
These Notes Have Python\_world\_In Notes Copy rights under copy right

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)

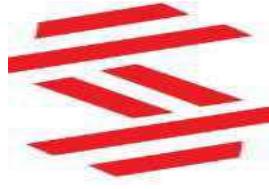


- As the name says, **Runtime Errors** are errors which occur while the **program** is **running**.
- As soon as **Python interpreter** encounters them it **halts the execution** of the program and **displays a message** about the probable cause of the problem.

# 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  
بُونی گُنْد جیسٹ 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" '''**

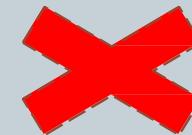
**OR**

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

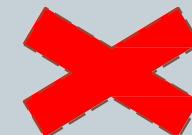
# Some Important Points About Strings



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



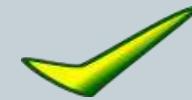
```
>>> msg="Let's learn "Python"""
      File "<stdin>", line 1
          msg="Let's learn "Python"
                  ^
SyntaxError: invalid syntax
```



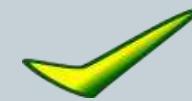
# Some Important Points About Strings



```
>>> 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**

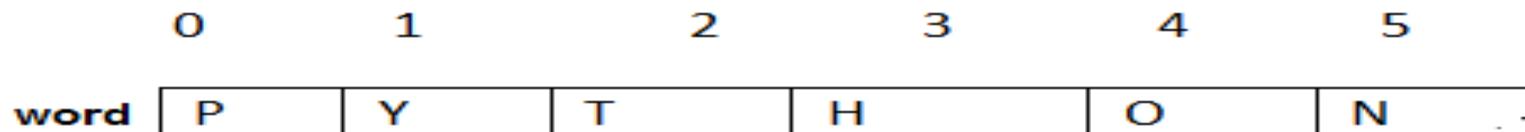
# Accessing Individual Characters In String



- 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



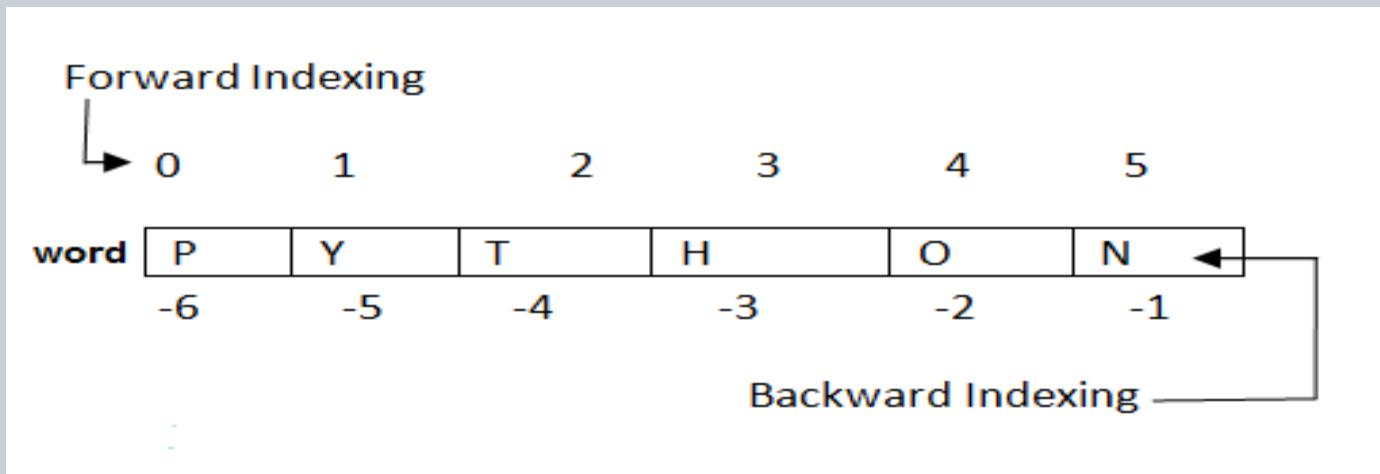
- However if we try to provide an **index number** beyond the given limit then **IndexError** exception will arise

```
>>> 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)`

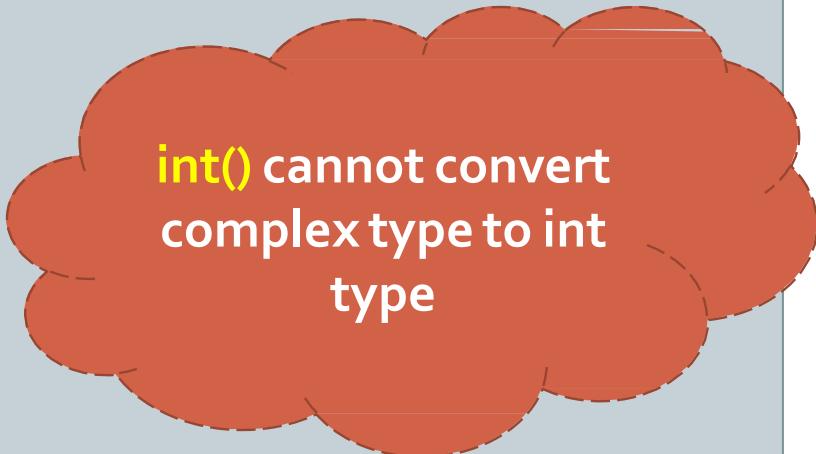
**Output:**

`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

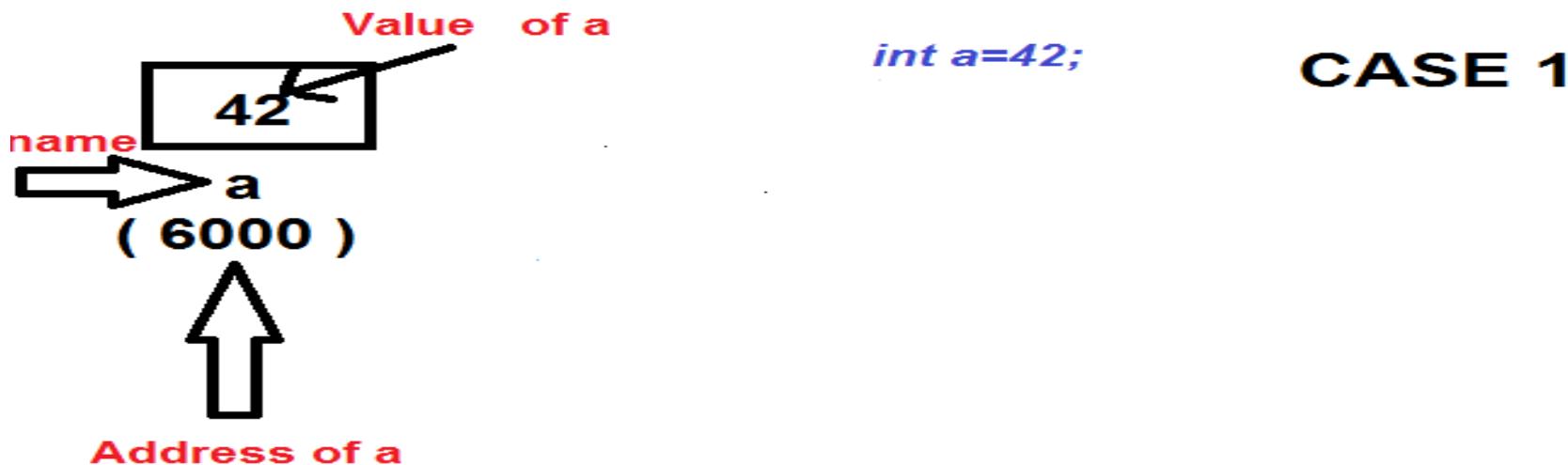


- A **variable** can be seen as a **container** to store certain values.
- While the program is running, **variables** are **accessed** and sometimes **changed**, i.e. a **new value** will be **assigned** to a **variable**

# How Variables Work In C ?



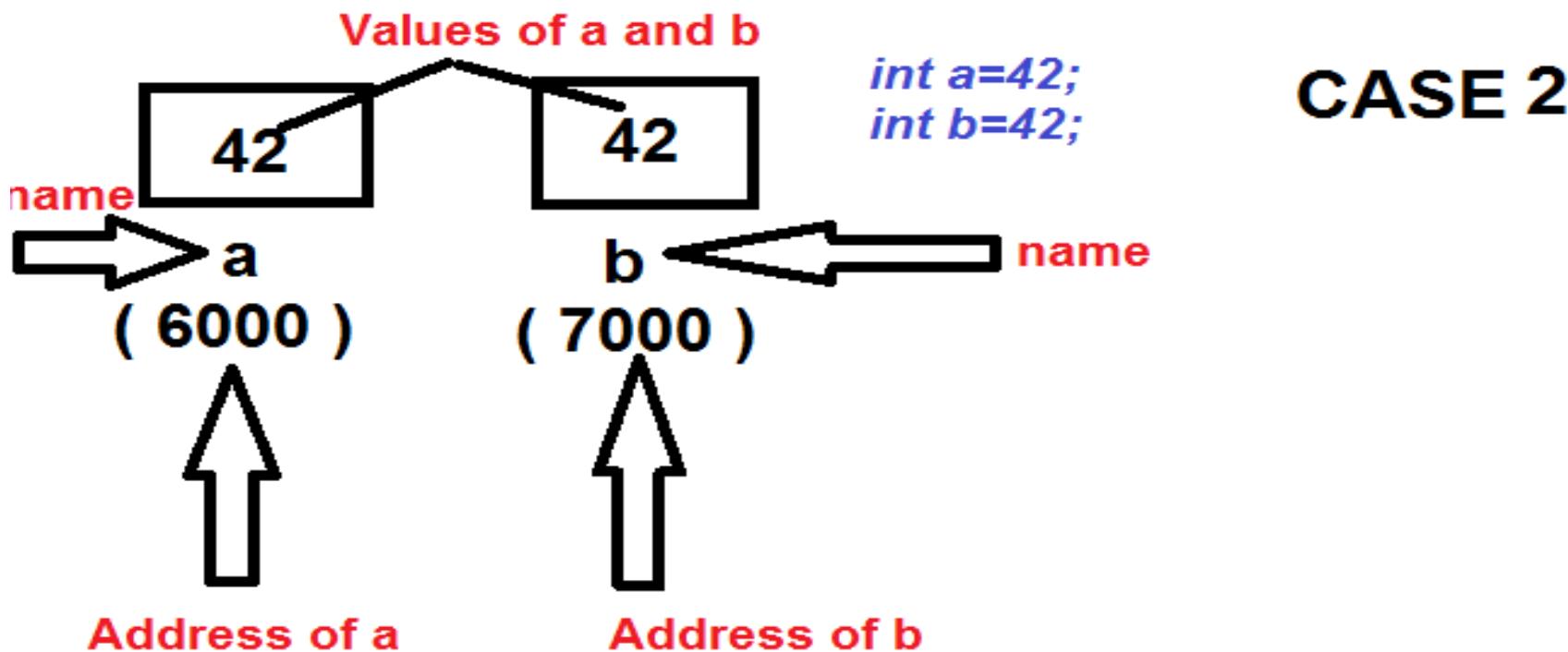
- In **C language** when we declare a **variable** and **assign** value to it then **some space is created** in memory by the **given name** and the **given value** is stored in it.
- Suppose we write the statement **int a=42;** , then the following will be the memory diagram.



# How Variables Work In C ?



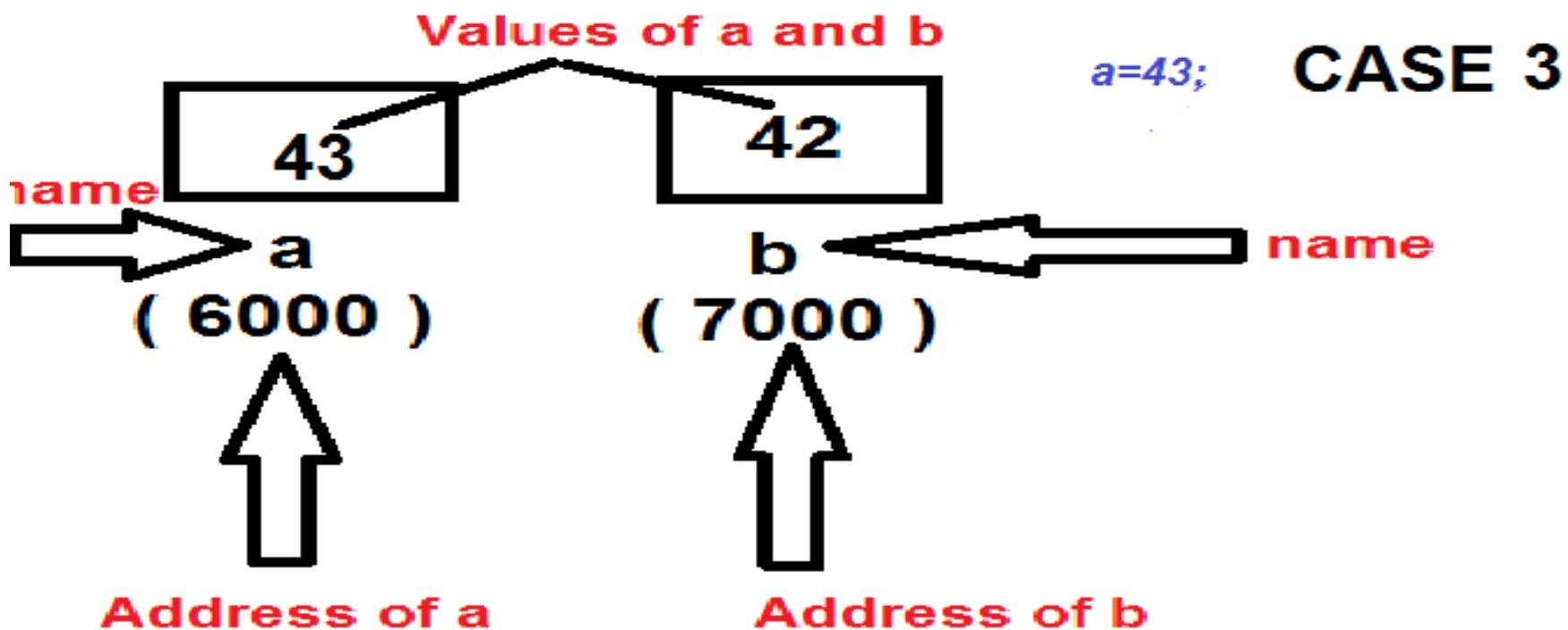
- Now if we declare **another variable**, with the **same value**, then again the **same process** will take place.
- Suppose we write , **int b=42;**



# 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 ?

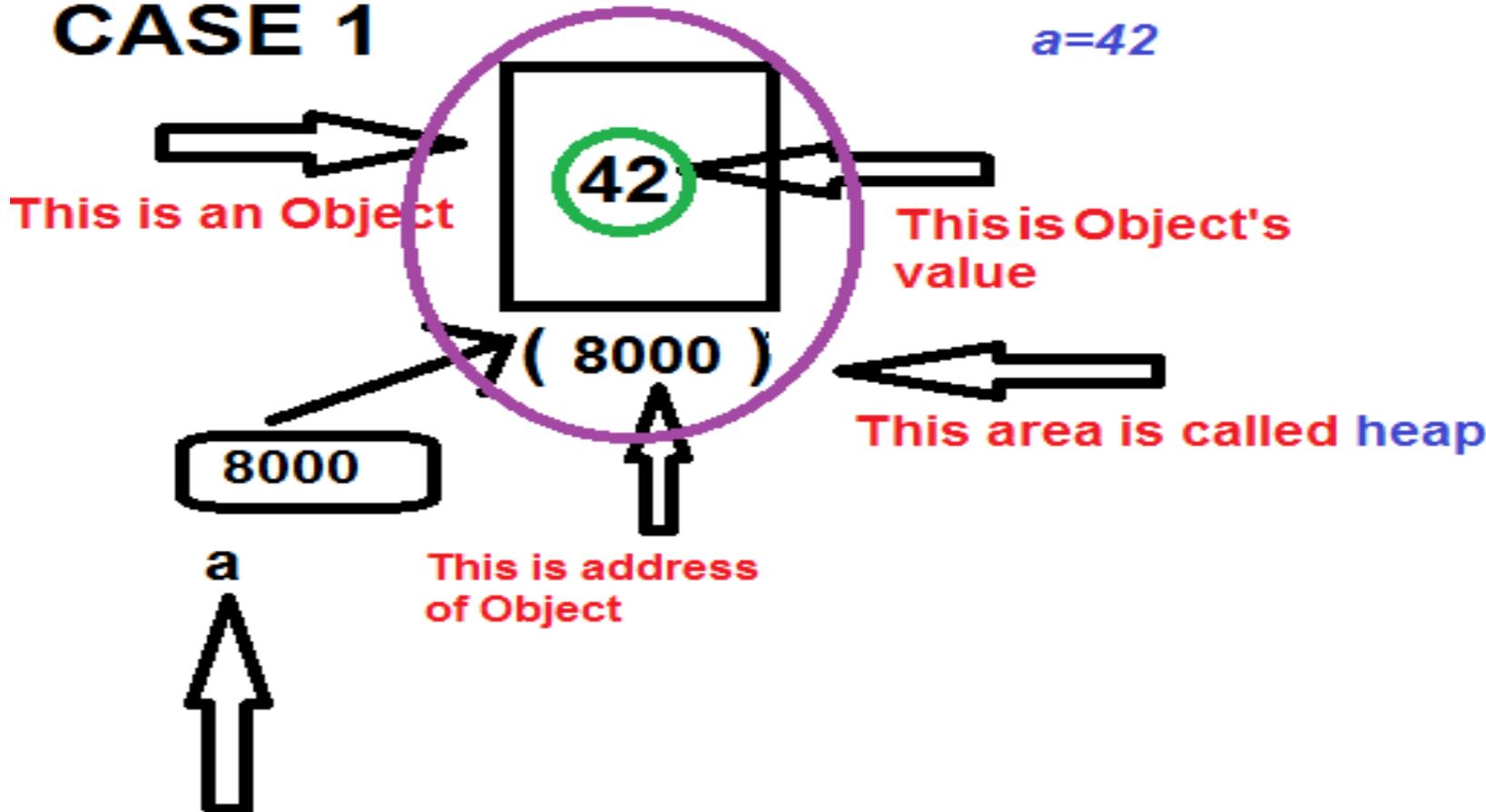


- In **Python** when we **assign** value to a **variable** , then things are **different** than **C** .
- Suppose we write **a=42** in **Python** , then **Python** will create 2 things:
  - An **object** in **heap memory** holding the **value 42** , and
  - A **reference** called **a** which will point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

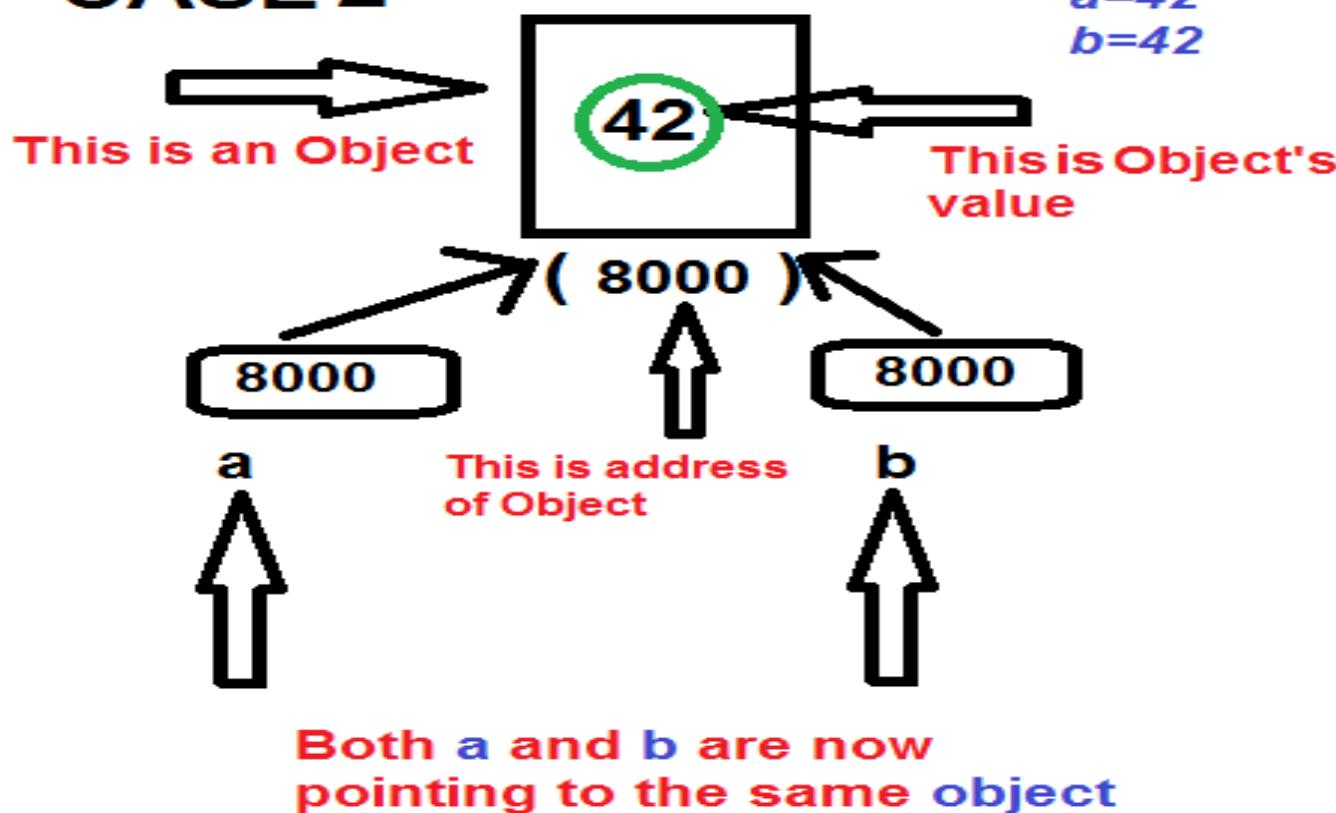


- Now if we **create another variable** called **b** and **assign** it the **same value**, then **Python** will do the following:
  - Create a new reference by the name b**
  - Assign the address of the previously created object to the reference b because the value is same**
  - So now both a and b are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

# How Variables Work In Python ?



## CASE 2



# How Variables Work In Python ?

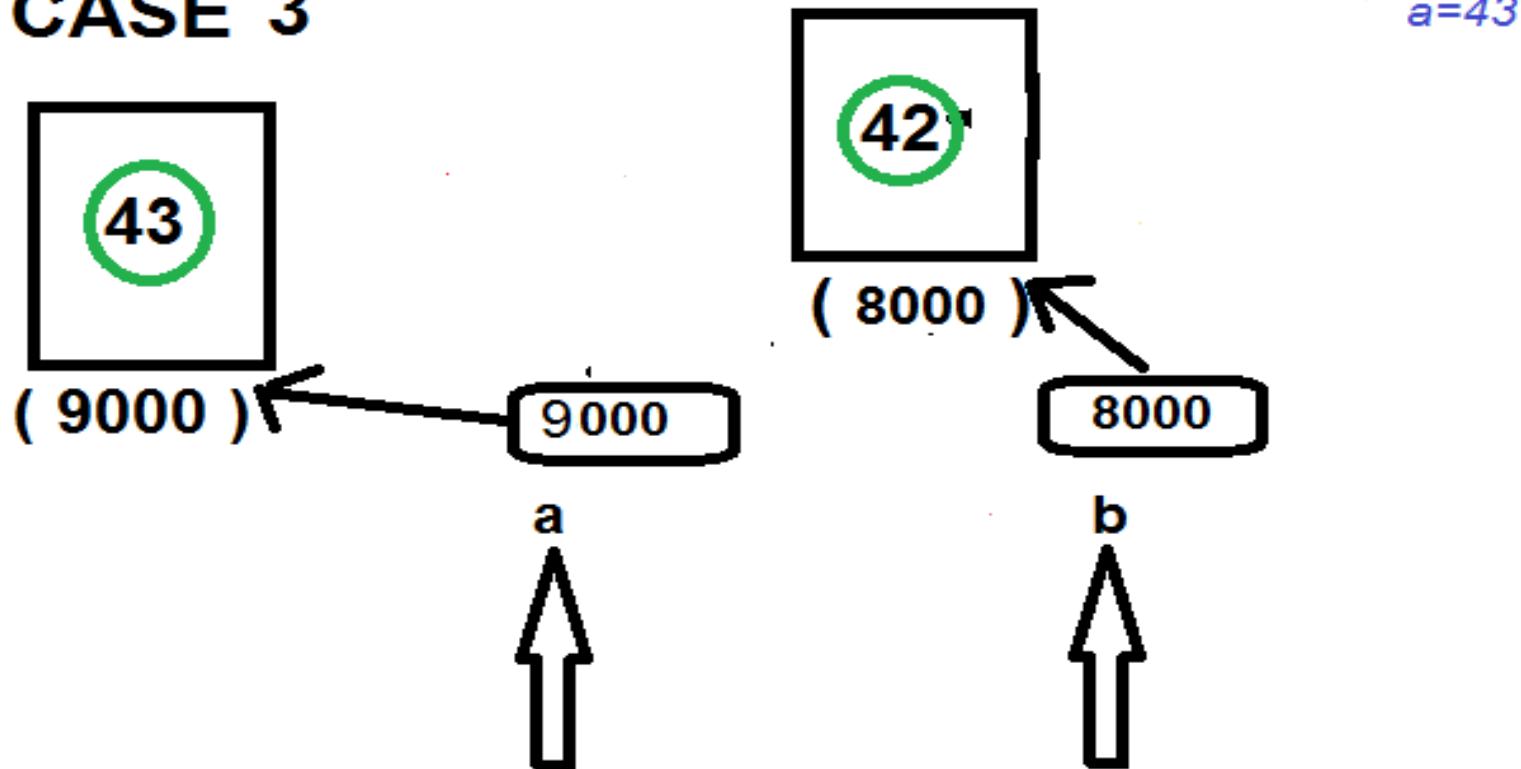


- Finally if we **assign** a new value to the variable **a** , then like **C** , **Python** will **not overwrite** the **value** .
- Rather it will do the following:
  - **Create a new object initialized with the new value**
  - **Assign the address of the newly created object to the reference a**
 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

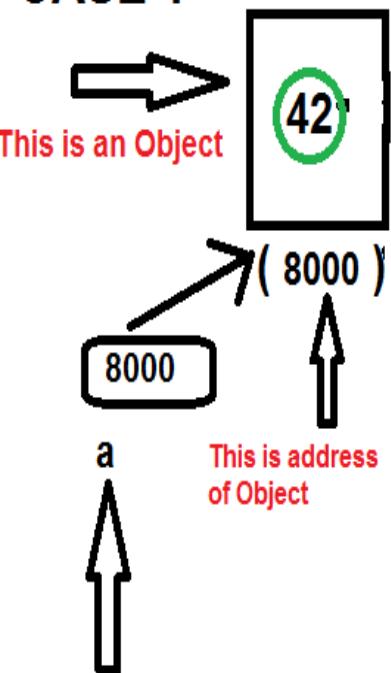


# The Garbage Block

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

# The Garbage Block

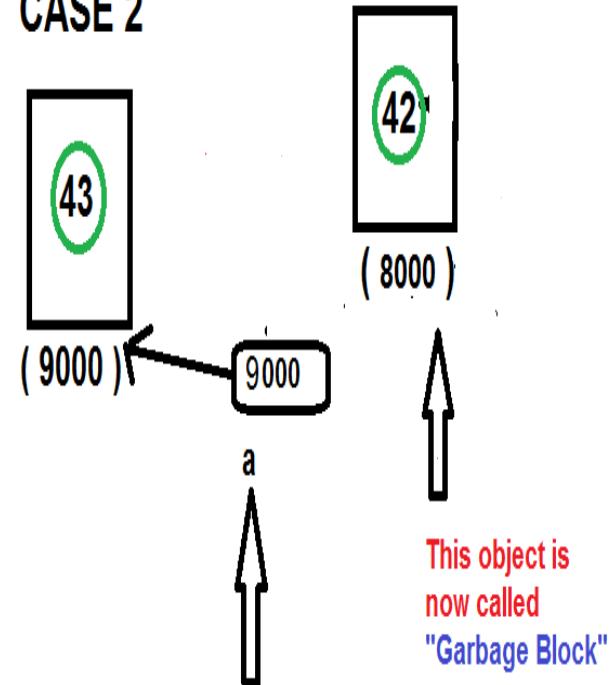
## CASE 1



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

**a=42**

## CASE 2



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**

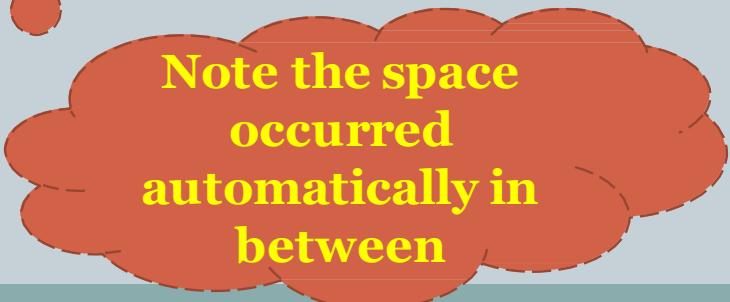
```
:
```

```
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="#")

- Example:

le:

hh=10

mm=30

ss=45

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

- Output:

10#20

- 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)
- Output:  
TypeError
- Example:  
a="Good"  
b="10"  
print(a+b)
- 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=20-12//3**2  
print(a)
```

Output:

19

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

Output:

1.0



# 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  $2\pi$

# 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

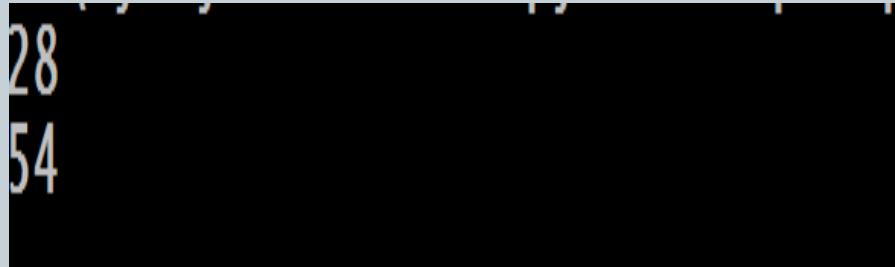


# Using WildCard Character



## 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 0<sup>th</sup> 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 **0<sup>th</sup>** 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



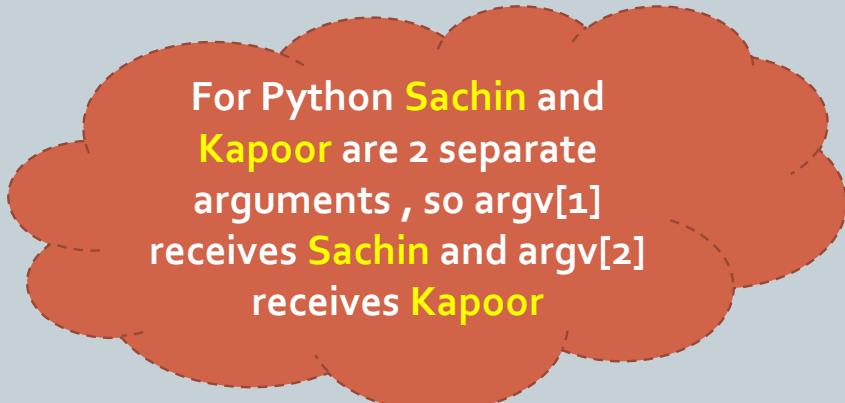
## Code:

```
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.

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("%d" %a)
```

□ Output:

1

```
a=True  
print("%f" %a)  
Output:  
1.000000
```



# 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