



# Conceptualizing Python in Google COLAB

**Dr. Poornima G. Naik**

**Dr. Girish R. Naik**

**Mr. M. B. Patil**

# **Conceptualizing Python in Google COLAB**

**By**

**Dr. Poornima G. Naik**

**Dr. Girish R. Naik**

**Mr. M.B.Patil**



**© Copyright 2021 Authors**

All rights are reserved. No part of this book may be reproduced or transmitted in any form by any means; electronic or mechanical including photocopy, recording, or any information storage or retrieval system; without the prior written consent of its author.

The opinions /contents expressed in this book are solely of the authors and do not represent the opinions / standings / thoughts of Shashwat Publication. No responsibility or liability is assumed by the publisher for any injury, damage or financial loss sustained to a person or property by the use of any information in this book, personal or otherwise, directly or indirectly. While every effort has been made to ensure reliability and accuracy of the information within, all liability, negligence or otherwise, by any use , misuse or abuse of the operation of any method, strategy, instruction or idea contained in the material herein is the sole responsibility of the reader. Any copyright not held by the publisher are owned by their respective authors. All information in this book is generalized and presented only for the informational purpose “as it is” without warranty or guarantee of any kind.

All trademarks and brands referred to in this book are only for illustrative purpose are the property of their respective owners and not affiliated with this publication in any way. The trademarks being used without permission don't authorize their association or sponsorship with this book.

**ISBN: 978-93-93557-43-8**

**Price: 250.00**

Publishing Year 2021

*Published and Printed by:*

**Shashwat Publication**

Office Address: Ram das Nagar,  
Bilaspur, Chhattisgarh – 495001

Phones: +91 9993608164 +91 9993603865

Email: [contact.shashwatpublication@gmail.com](mailto:contact.shashwatpublication@gmail.com)

Website: [www.shashwatpublication.com](http://www.shashwatpublication.com)

Printed in India



## Acknowledgements

Many individuals share credit for this book's preparation. We extend our sincere thanks to Late Prof. A.D.Shinde, the Founder Director and Managing Trustee who has been a constant source of inspiration for us throughout our career. His support is really a driving force for us. Also, we would like to thank Dr.R.A.Shinde, Hon'ble Secretary, CSIBER, Kolhapur for his whole hearted support and continuous encouragement. We are grateful to Dr. C.S.Dalvi, Director CSIBER, Kolhapur for his invaluable guidance. We take this opportunity to thank Dr.V.M.Hilage, Former Director and Trustee Member, CSIBER, Kolhapur, Mr. Bharat Patil, Chairman, Mr. Sunil Kulkarni, Vice Chairman, and Deepak Chougule, Secretary, KIT, Kolhapur for showing a keen interest in the matter of this book and extending all support facilities for the in-timely completion of this book. Last but not the least we thank all faculty members and non-teaching staff of department of computer studies, CSIBER, Kolhapur and Production Engineering department, KIT's College of Engineering, Kolhapur who have made contribution to this book either directly or indirectly.

**Dr. Poornima G. Naik**

**Dr. Girish R. Naik**

**Mr. M.B.Patil**

## Preface

It gives us an immense pleasure to bring out a book entitled '***Conceptualizing Python in Google COLAB***'. The goal of this book is to introduce you to the topic and get you started on your journey to application development using Python. There are many books, websites, online courses, tutorials etc. on Python. The current book is different in that it does not provide a lengthy tutorial introduction to a particular aspect Python, but gives the practical inputs for learning Python in a simple and effective way. The book offers first hand acquaintance with the Python language and the new constructs added to the language in order to render it safe, clean and robust. It aims to provide comprehensive material on Python.

### **How is this Book Organized:**

This book can serve as textbook for post graduates and reference for any computer graduate. It will also provide easy reference for Computer Professionals who wants to begin their career in Machine Learning using Python. This book is precisely organized into twelve chapters. Each chapter has been carefully developed with the help of several implemented concepts. Dedicated efforts have been put in to ensure that every concept of Python discussed in this book is explained with help of relevant commands and screenshots of the outputs have been included. **Chapter 1** focuses on development environment offered by Google COLAB. **Chapters 2 through 4** cover the Python language fundamentals focusing on control and iterative statements, operators along with their applications in basic programs. Python employs blended programming paradigm in which it is procedural, object-oriented and functional. The best part of all programming languages reside in a single platform. **Chapter 5** focuses on functions in Python with a special emphasis on Lambda functions. Advanced Python programming concepts such as iterators, closures, decorators, generators are covered at depth in **Chapter 6 and 7**. A good and in-depth knowledge of exception handling enables in writing a reliable and robust code. To cater to this need **Chapter 8** unleashes the salient features of exception handling in Python. Data persistence through file handling is covered in **Chapter 9**. Due to the wide application of Regular expressions in pattern matching, **Chapter 10** is fully devoted to understanding of regular expression in Python. Different types of common errors that might creep in during the execution of a Python program are summarized in **Chapter 11**. Final **Chapter 12** is devoted to implementation of object oriented concepts in Python. The case study based on object oriented concept is discussed at depth and implemented in **Appendix A**.

The part of the content of this book is derived from different sources which are listed at the end in a '*references*' section.

**Dr. Poornima G. Naik**

**Dr. Girish R. Naik**

**Mr. M.B.Patil**

# Contents

<b>Chapter</b>	<b>Page No.</b>
1. Introduction to Google COLAB	1
2. Lab Assignments on Python Language Fundamentals	33
3. Lab Assignment on Python Operators and Control Statements	102
4. Lab Assignment on Basic Programs	117
5. Lab Assignment on Python Functions	130
6. Lab Assignment on Advanced Concepts in Python - I (Covers Iterators, Closures, Decorators and Generators)	142
7. Lab Assignment on Advanced Concepts in Python - II	157
8. Lab Assignment on Exception Handling in Python	170
9. Lab Assignment on File Handling in Python	204
10. Lab Assignment on Regular Expressions in Python	228
11. Lab Assignment on Language Basics and Error Handling in Python	251
12. Lab Assignment on Object Oriented Programming in Python	257
References	303
Appendix A - Case Study on Object Oriented Programming	304

---

v

# Conceptualizing Python in Google COLAB

---

## Chapter 1

### Introduction to Google COLAB



Colaboratory, or '**Colab**' for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education.

If you are exploring Machine Learning but struggling to conduct simulations on enormous datasets, or an expert playing with ML desperate for extra computational power, Google Colab is the perfect solution for you. Google Colab or '**the Colaboratory**' is a free cloud service hosted by Google to encourage Machine Learning and Artificial Intelligence research, where often the barrier to learning and success is the requirement of tremendous computational power.

If you want to create a machine learning model but you don't have a computer that can take the workload, Google Colab is the platform for you. Even if you have a GPU or a good computer creating a local environment with anaconda and installing packages and resolving installation issues are a hassle.

Colaboratory is a free Jupyter notebook environment provided by Google where you can use free GPUs and TPUs which can solve all these issues. It contains almost all the modules you need for data science analysis. These tools include but are not limited to Numpy, Scipy, Pandas, etc. Even deep learning frameworks, such as Tensorflow, Keras and Pytorch are also included.

### Benefits of COLAB

Colab is a free Jupyter notebook environment that runs entirely in the cloud.

Besides being easy to use, the Colab is fairly flexible in its configuration and does much of the heavy lifting for you. It does not require a setup.

- Python 2.7 and Python 3.6 support

# Conceptualizing Python in Google COLAB

---

- Free GPU acceleration
- Pre-installed libraries: All major Python libraries like TensorFlow, Scikit-learn, Matplotlib among many others are pre-installed and ready to be imported.
- Built on top of Jupyter Notebook
- Collaboration feature (works with a team just like Google Docs): Google Colab allows developers to use and share Jupyter notebook among each other without having to download, install, or run anything other than a browser. Notebooks can be shared among team members.
- Supports bash commands
- Google Colab notebooks are stored on the drive

## What COLAB Offers You?

As a programmer, you can perform the following using Google Colab.

- Write and execute code in Python
- Document your code that supports mathematical equations
- Create/Upload/Share notebooks
- Import/Save notebooks from/to Google Drive
- Import/Publish notebooks from GitHub
- Import external datasets e.g. from Kaggle
- Integrate PyTorch, TensorFlow, Keras, OpenCV
- Free Cloud service with free GPU

## Getting Started with COLAB

To start working with Colab you first need to log in to your Google account, then go to this link

<https://colab.research.google.com>.

Since Colab implicitly uses Google Drive for storing your notebooks, ensure that you are logged in to your Google Drive account before proceeding further.

# Conceptualizing Python in Google COLAB

---



## What is Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

## Opening Jupyter Notebook:

On opening the website you will see a pop-up containing following tabs –

A screenshot of the Jupyter Notebook interface showing the 'Recent' tab selected. The top navigation bar includes 'Examples', 'Recent', 'Google Drive', 'GitHub', and 'Upload'. Below the navigation is a search bar labeled 'Filter notebooks'. A table lists recent notebooks: 'Welcome To Colaboratory' (last opened 7:17 PM, first opened 6:59 PM) and 'test.ipynb' (last opened 6:59 PM, first opened 6:59 PM). Each notebook entry has a delete icon and a refresh icon. At the bottom right are 'NEW NOTEBOOK' and 'CANCEL' buttons.

Title	Last opened	First opened	Actions
Welcome To Colaboratory	7:17 PM	6:59 PM	
test.ipynb	6:59 PM	6:59 PM	

NEW NOTEBOOK CANCEL

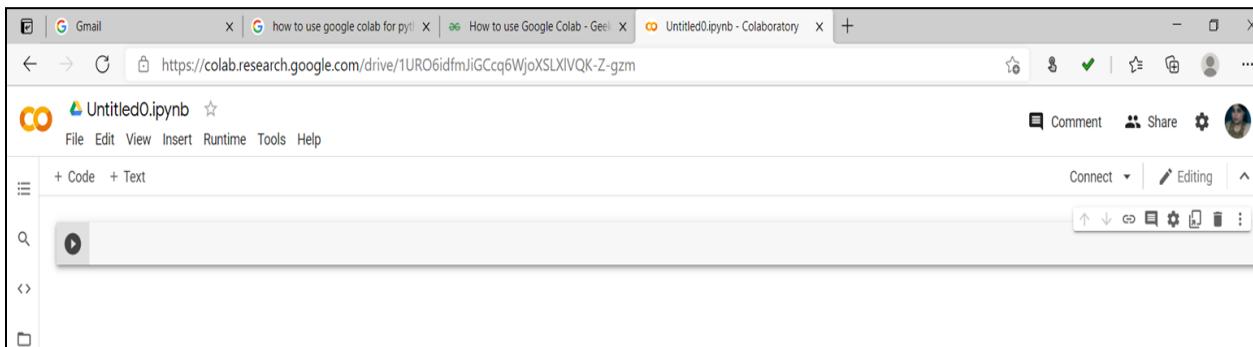
# Conceptualizing Python in Google COLAB

On opening the website you will see a pop-up containing following tabs –

**EXAMPLES:** Contain a number of Jupyter notebooks of various examples.  
**RECENT:** Jupyter notebook you have recently worked with.  
**GOOGLE DRIVE:** Jupyter notebook in your google drive.  
**GITHUB:** You can add Jupyter notebook from your GitHub but you first need to connect Colab with GitHub.  
**UPLOAD:** Upload from your local directory.

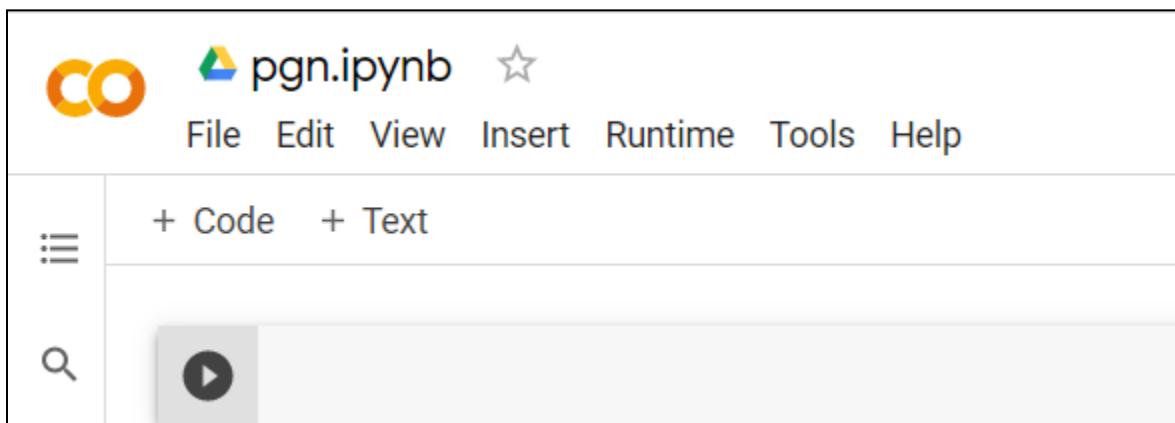
On opening the website you will see a pop-up containing following tabs –

Else you can *create a new Jupyter notebook* by clicking New Python3 Notebook or New Python2 Notebook at the bottom right corner.



On creating a new notebook, it will create a Jupyter notebook with Untitled0.ipynb and save it to your Google drive in a folder named '**Colab Notebooks**'.

You can click in the file name and change it as shown below. The extension of the file is **.ipynb**.



# Conceptualizing Python in Google COLAB

---

## Entering Code

You will now enter a trivial Python code in the code window and execute it.

Enter the following two Python statements in the code window –

*import time*

*print(time.ctime())*

## Executing Code

To execute the code, click on the arrow on the left side of the code window.

A screenshot of a Google Colab notebook cell. On the left, there is a play button icon. The code area contains the following Python code:

```
import time
print(time.ctime())
```

The output area shows the result of the execution:

Fri May 28 05:17:38 2021

## Clearing Output

You can clear the output anytime by clicking the icon on the left side of the output display.

A screenshot of a Google Colab notebook cell. On the left, there is a play button icon. The code area contains the same Python code as before. The output area shows the result of the execution:

Fri May 28 05:17:38 2021

Below the output, a context menu is open with the following options:

- Clear output
- executed by Poornima Naik
- 10:47 AM (7 minutes ago)
- executed in 0.023s

## Adding New Cells

To add more code to your notebook, select the following **menu** options –

---

# Conceptualizing Python in Google COLAB

---

## **Insert / Code Cell**

Alternatively, just hover the mouse at the bottom center of the Code cell. When the ‘**Code**’ and ‘**Text**’ buttons appear, click on the ‘**Code**’ to add a new cell. This is shown in the screenshot below –

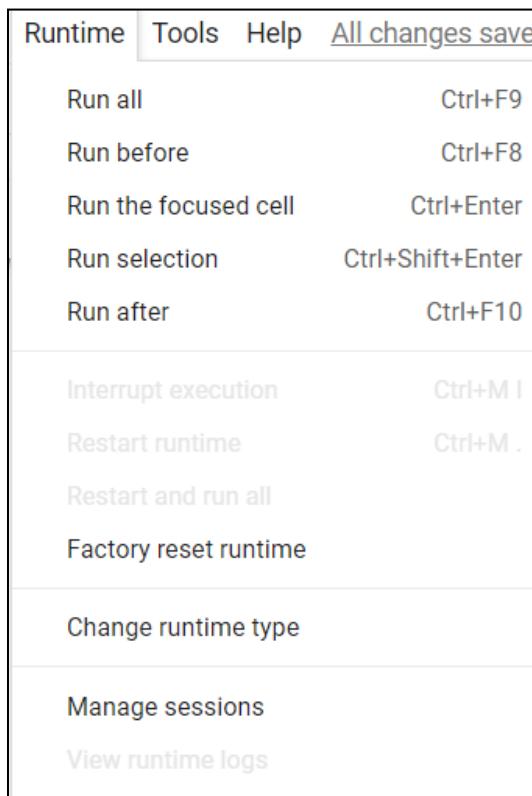
```
import time
print(time.ctime())
|
```

Fri May 28 05:17:38 2021

+ Code + Text

## Changing Runtime Environment:

Click the ‘**Runtime**’ dropdown menu. Select ‘**Change runtime type**’. Select python2 or 3 from ‘**Runtime type**’ dropdown menu.

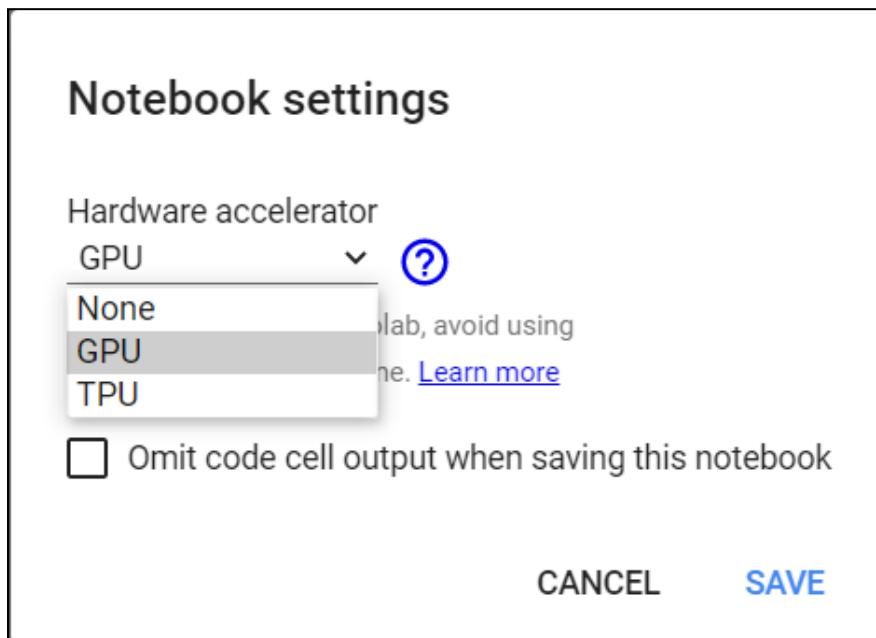


# Conceptualizing Python in Google COLAB

---

## What is GPU?

Graphics processing unit, a specialized processor originally designed to accelerate graphics rendering. **GPUs** can process many pieces of data simultaneously, making them useful for machine learning, video editing, and gaming applications. Select '**Change runtime type**'. Colab provides the **Tesla K80 GPU**.



## Verifying GPU

Enter the following code in the cell and execute.

```
import tensorflow as tf  
tf.test.gpu_device_name()
```

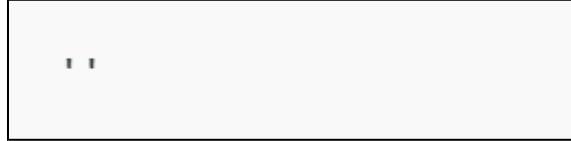
- If gpu is connected it will output following –

```
'/device:GPU:0'
```

- Otherwise, it will output following

# Conceptualizing Python in Google COLAB

---



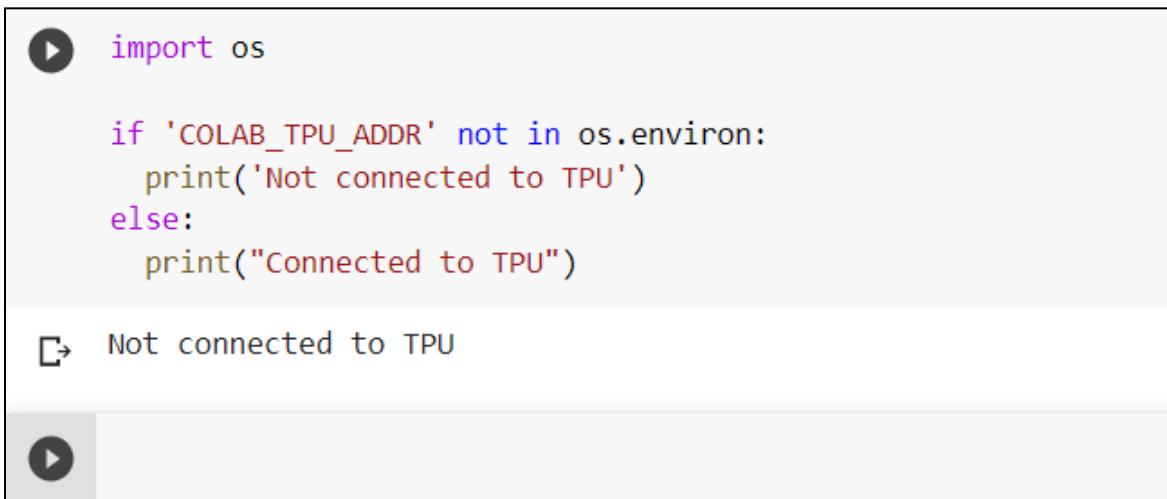
## What is TPU?

**TPUs** are tensor processing units developed by Google to accelerate operations on a Tensorflow Graph. Each **TPU** packs up to 180 teraflops of floating-point performance and 64 GB of high-bandwidth memory onto a single board.

## Verifying TPU

Enter the following code in the cell and execute.

```
import os  
  
if 'COLAB_TPU_ADDR' not in os.environ:  
    print('Not connected to TPU')  
  
else:  
    print("Connected to TPU")
```



```
import os  
  
if 'COLAB_TPU_ADDR' not in os.environ:  
    print('Not connected to TPU')  
else:  
    print("Connected to TPU")
```

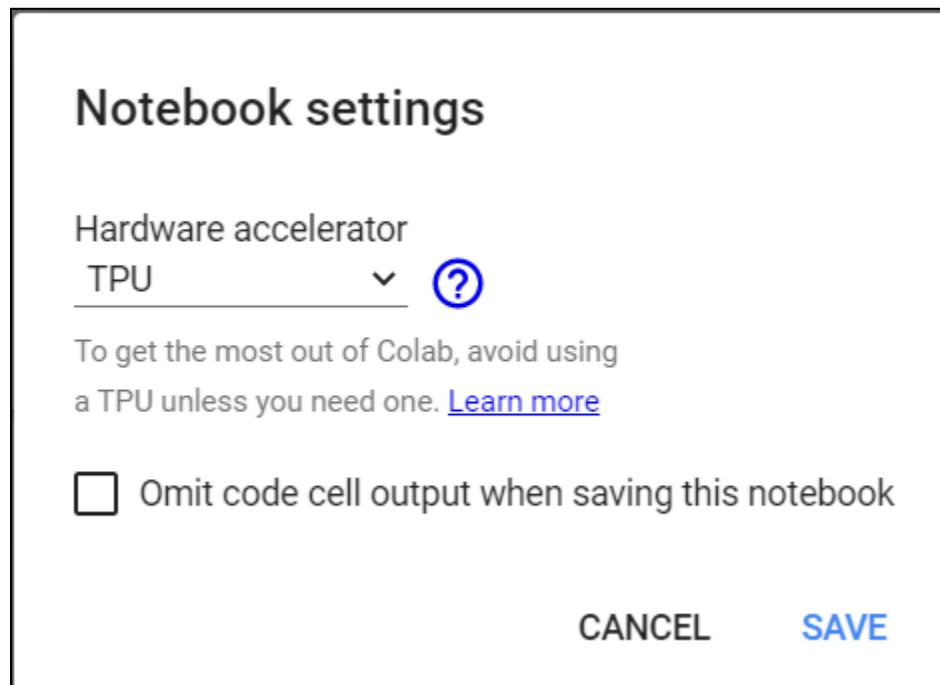
Not connected to TPU

## Verifying TPU

Select the option **Runtime → Change Runtime Type** and select ‘**TPU**’ from the ‘**Hardware Selector**’ dropdown list and re-execute the program.

# Conceptualizing Python in Google COLAB

---



## Verifying TPU

The following output is displayed.

```
[1] import os

if 'COLAB_TPU_ADDR' not in os.environ:
    print('Not connected to TPU')
else:
    print("Connected to TPU")
```

Connected to TPU

## Installing Python packages –

You can use '**pip**' to install any package. For example:

For installing a package '**pandas**' enter the following command:

**pip install pandas**

# Conceptualizing Python in Google COLAB

The following output is displayed:

```
[2] pip install pandas

Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (1.1.5)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-packages (from pandas) (2018.9)
Requirement already satisfied: numpy>=1.15.4 in /usr/local/lib/python3.7/dist-packages (from pandas) (1.19.5)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist-packages (from pandas) (2.8.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.7.3->pandas) (1.15.0)
```

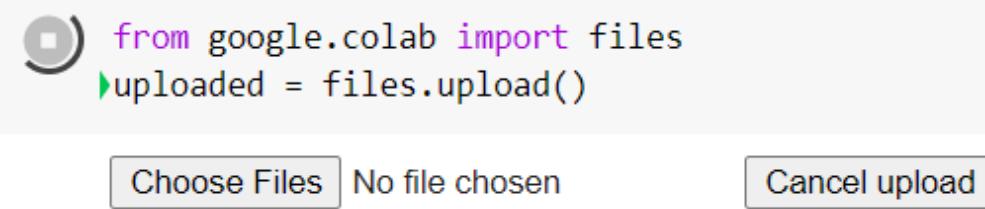
## Uploading a File in COLAB

For uploading a file, enter the following code:

```
from google.colab import files
```

```
uploaded = files.upload()
```

‘Choose Files’ button appears as shown below:



On selecting the file, the file is uploaded on the drive and the following output is generated:

```
[3] from google.colab import files
uploaded = files.upload()

Choose Files admission_sheet.xlsx
• admission_sheet.xlsx(application/vnd.openxmlformats-officedocument.spreadsheetml.sheet) - 10689 bytes, last modified: 7/29/2020 - 100% done
Saving admission_sheet.xlsx to admission_sheet.xlsx
```

## Mounting Drive

- Unix systems have a single directory tree. All accessible storage must have an associated location in this single directory tree. This is unlike Windows where (in the most common syntax for file paths) there is one directory tree per storage component (drive).
- Mounting is the act of associating a storage device to a particular location in the directory tree. For example, when the system boots, a particular storage device (commonly called

# Conceptualizing Python in Google COLAB

---

the root partition) is associated with the root of the directory tree, i.e., that storage device is mounted on / (the root directory).

- Let's say you now want to access files on a CD-ROM. You must mount the CD-ROM on a location in the directory tree (this may be done automatically when you insert the CD). Let's say the CD-ROM device is /dev/cdrom and the chosen mount point is /media/cdrom. The corresponding command is

***mount /dev/cdrom /media/cdrom***

- After that command is run, a file whose location on the CD-ROM is /dir/file is now accessible on your system as ***/media/cdrom/dir/file***. When you've finished using the CD, you run the command

***umount /dev/cdrom or umount /media/cdrom***

(both will work; typical desktop environments will do this when you click on the '**eject**' or '**safely remove**' button).

- Mounting applies to anything that is made accessible as files, not just actual storage devices. For example, all Linux systems have a special file system mounted under /proc. That file system (called proc) does not have underlying storage: the files in it give information about running processes and various other system information; the information is provided directly by the kernel from its in-memory data structures.

## Uploading a File:

The uploaded file can be saved in a data frame as shown below:

```
import io  
df2 = pd.read_csv(io.BytesIO(uploaded['file_name.csv']))
```

## Upload File By Mounting Google Drive:

To mount your drive inside '***mntDrive***' folder execute following –

```
from google.colab import drive  
drive.mount('/mntDrive')
```

# Conceptualizing Python in Google COLAB

---

```
● from google.colab import drive  
drive.mount('/mntDrive')  
  
Drive already mounted at /mntDrive; to attempt to forcibly remount, call drive.mount("/mntDrive", force_remount=True).
```

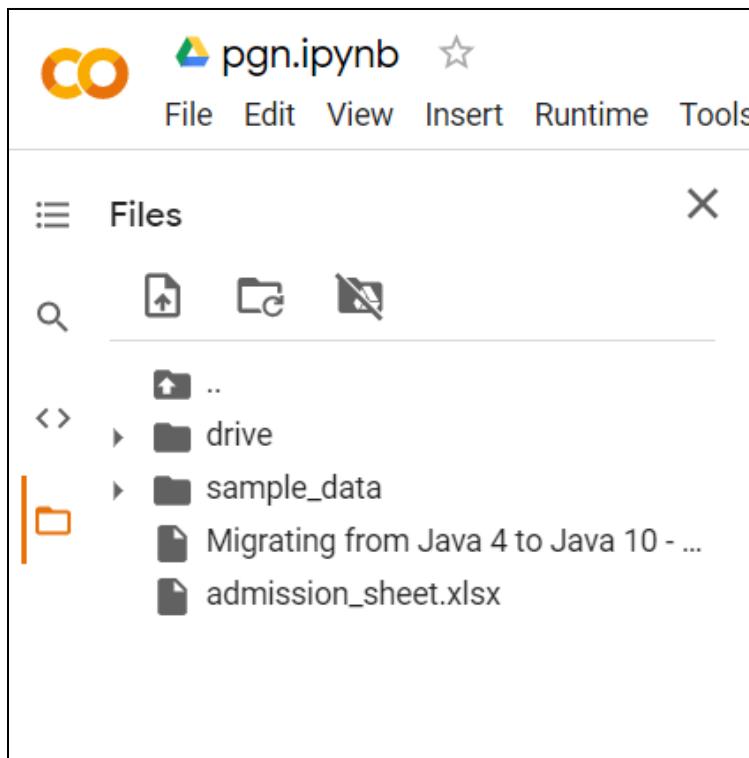
## Uploading a File:

Then you'll see a link, click on link, then allow access, copy the code that pops up, paste it at "Enter your authorization code:".

Now to see all data in your Google drive you need to execute following:

***! ls "/mntDrive/My Drive"***

Mounting Drive Visually



## Running a Cell

Make sure the runtime is connected. The notebook shows a green check and '***Connected***' on the top right corner.

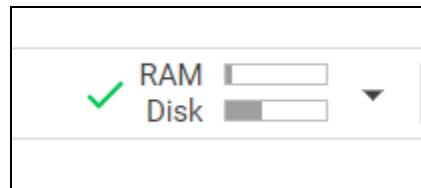
There are various runtime options in '***Runtime***'.

OR

# Conceptualizing Python in Google COLAB

---

To run the current cell, press SHIFT + ENTER.



## Running a Cell

```
!cat /proc/cpuinfo
```

```
!cat /proc/meminfo
```

```
!cat /proc/cpuinfo
!cat /proc/meminfo

processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 85
model name    : Intel(R) Xeon(R) CPU @ 2.00GHz
stepping       : 3
microcode     : 0x1
cpu MHz       : 2000.174
cache size    : 39424 KB
physical id   : 0
siblings       : 2
core id        : 0
cpu cores     : 1
apicid         : 0
initial apicid: 0
fpu            : yes
fpu_exception  : yes
cpuid level   : 13
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
bugs           : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs
bogomips       : 4000.34
clflush size   : 64
cache_alignment : 64
address sizes  : 46 bits physical, 48 bits virtual
power management:
```

## Saving to Google Drive

Colab allows you to save your work to your Google Drive. To save your notebook, select the following menu options –

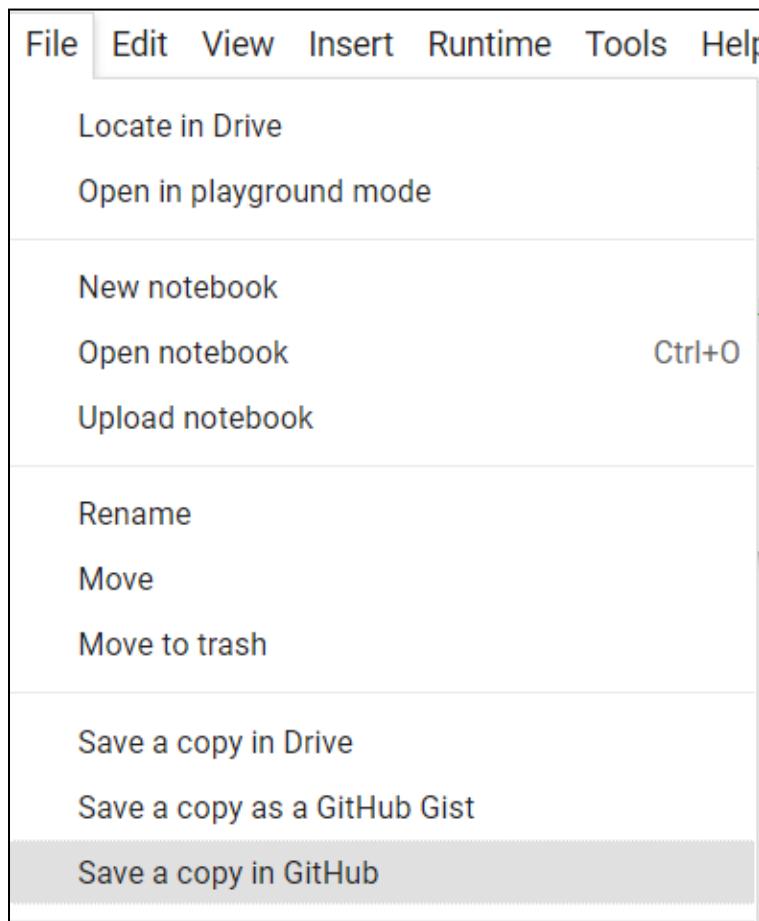
# Conceptualizing Python in Google COLAB

---

## *File / Save a copy in Drive...*

You will see the following screen –

The action will create a copy of your notebook and save it to your drive. Later on you may rename the copy to your choice of name.



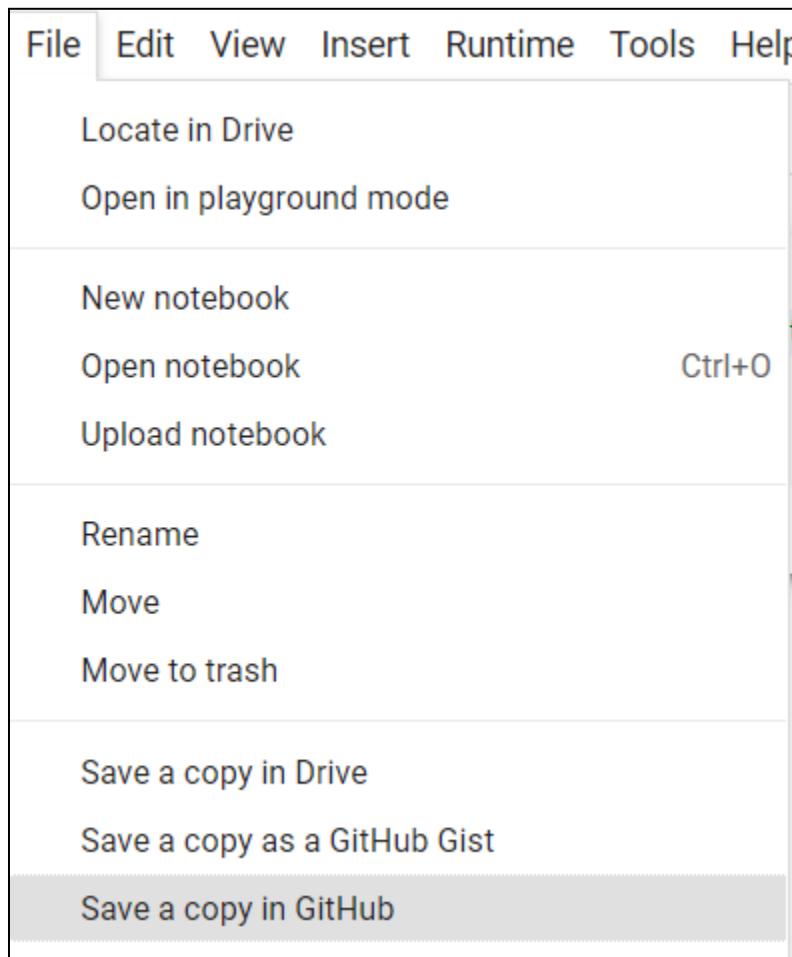
You may also save your work to your GitHub repository by selecting the following menu options –

## *File / Save a copy in GitHub...*

The menu selection is shown in the following screenshot for your quick reference –

# Conceptualizing Python in Google COLAB

---



You will have to wait until you see the login screen to GitHub. Now, enter your credentials. If you do not have a repository, create a new one and save your project

## Documenting Your Code

- As the code cell supports full Python syntax, you may use Python **comments** in the code window to describe your code. However, many a time you need more than a simple text based comments to illustrate the ML algorithms. ML heavily uses mathematics and to explain those terms and equations to your readers you need an editor that supports **LaTeX** - *a language for mathematical representations*. Colab provides **Text Cells** for this purpose.
- A text cell containing few mathematical equations typically used in ML is shown in the screenshot below –

# Conceptualizing Python in Google COLAB

---

Text Cells are formatted using **markdown** - a simple markup language. Let us now see you how to add text cells to your notebook and add to it some text containing mathematical equations.

## Markdown Examples

Let us look into few examples of markup language syntax to demonstrate its capabilities.

Type in the following text in the Text cell.

*This is \*\*bold\*\*.*

*This is \*italic\*.*

*This is ~strikethrough~.*

The screenshot shows a text cell in Google Colab. The toolbar at the top includes icons for text style (T), bold (B), italic (I), code (C), image (Image), list (List), table (Table), and more. The text area contains three lines of code: "This is \*\*bold\*\*.", "This is \*italic\*.", and "This is ~strikethrough~.". Below the text area, a larger box displays the rendered output: "This is bold. This is italic. This is strikethrough."

## Mathematical Equations

Add a Text Cell to your notebook and enter the following markdown syntax in the text window –

$\$|sqrt{3x-1}+(1+x)^2\$$

You will see the immediate rendering of the markdown code in the right hand side panel of the text cell. This is shown in the screenshot below –

The screenshot shows a text cell in Google Colab. The toolbar at the top is identical to the previous one. The text area contains the markdown code: "\$\\sqrt{3x-1} + (1+x)^2\$". Below the text area, a larger box displays the rendered output:  $\sqrt{3x - 1} + (1 + x)^2$ .

Hit **Enter** and the markdown code disappears from the text cell and only the rendered output is shown

# Conceptualizing Python in Google COLAB

Let us try another more complicated equation as shown here –

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!} x^i$$

The rendered output is shown here for your quick reference.

The screenshot shows a Google Colab notebook cell. The cell contains the following text:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!} x^i$$

Constraints are

- $3x_1 + 6x_2 + x_3 \leq 28$
- $7x_1 + 3x_2 + 2x_3 \leq 37$
- $4x_1 + 5x_2 + 2x_3 \leq 19$
- $x_1, x_2, x_3 \geq 0$

The screenshot shows a Google Colab notebook cell. The cell contains the following text:

Constraints are

- $3x_1 + 6x_2 + x_3 \leq 28$
- $7x_1 + 3x_2 + 2x_3 \leq 37$
- $4x_1 + 5x_2 + 2x_3 \leq 19$
- $x_1, x_2, x_3 \geq 0$

# Conceptualizing Python in Google COLAB

## What is LaTex?

LaTeX is a document preparation system and document markup language. LaTeX is not the name of a particular editing program, but refers to the encoding or tagging conventions that are used in LaTeX documents.

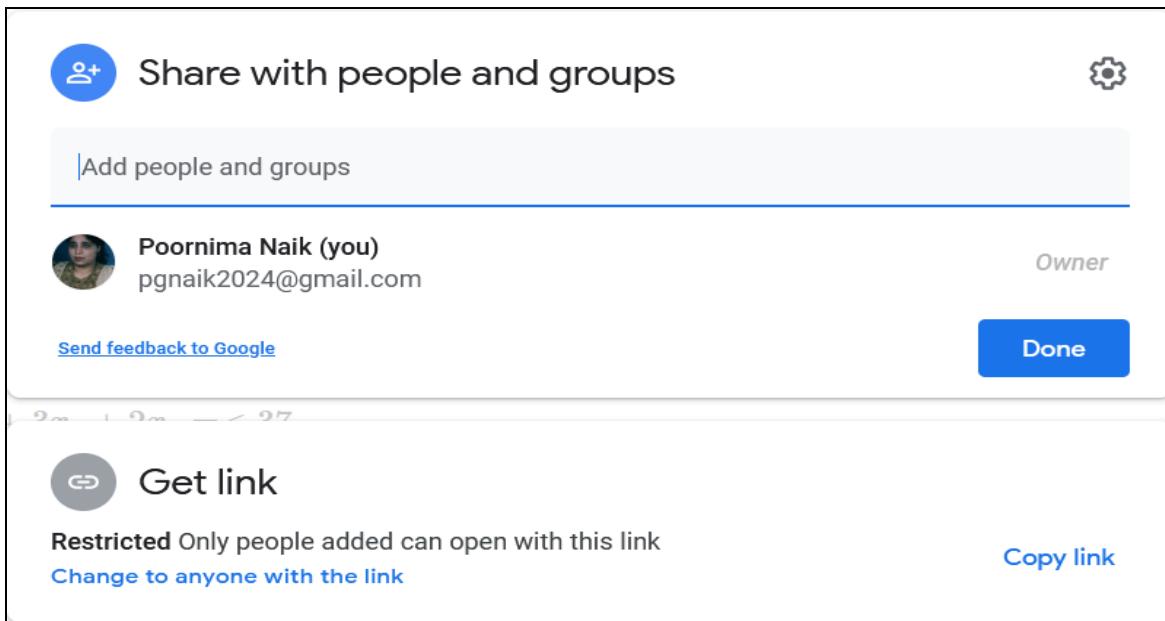
### LATEX math constructs

$\frac{abc}{xyz}$	<code>\frac{abc}{xyz}</code>	$\overline{abc}$	<code>\overline{abc}</code>	$\overrightarrow{abc}$	<code>\overrightarrow{abc}</code>
$f'$	<code>f'</code>	$\underline{abc}$	<code>\underline{abc}</code>	$\overleftarrow{abc}$	<code>\overleftarrow{abc}</code>
$\sqrt{abc}$	<code>\sqrt{abc}</code>	$\widehat{abc}$	<code>\widehat{abc}</code>	$\overbrace{abc}$	<code>\overbrace{abc}</code>
$\sqrt[n]{abc}$	<code>\sqrt[n]{abc}</code>	$\widetilde{abc}$	<code>\widetilde{abc}</code>	$\underbrace{abc}$	<code>\underbrace{abc}</code>

## Google Colab - Sharing Notebook

- To share the notebook that you have created with other co-developers, you may share the copy that you have made in your Google Drive.
- To publish the notebook to general audience, you may share it from your GitHub repository.

There is one more way to share your work and that is by clicking on the **SHARE** link at the top right hand corner of your Colab notebook. This will open the share box as shown here



# Conceptualizing Python in Google COLAB

---

You may enter the email IDs of people with whom you would like to share the current document.

You can set the kind of access by selecting from the three options shown in the above screen.

Click on the Get shareable link option to get the URL of your notebook. You will find options for whom to share as follows –

- Specified group of people
- Colleagues in your organization
- Anyone with the link
- All public on the web

Now, you know how to create/execute/save/share a notebook. In the Code cell, we used Python so far.

## Getting Remote Data

Let us look into another example that loads the dataset from a remote server.

Type in the following command in your Code cell –

```
!wget http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.data -P  
"/content/drive/My Drive/app"
```

## Cloning Git Repository

You can clone the entire GitHub repository into Colab using the **git** command. For example, to clone the keras tutorial, type the following command in the Code cell –

```
!git clone https://github.com/wxs/keras-mnist-tutorial.git
```

## What is Git?

*Git* is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

## Google Colab - Graphical Outputs

Colab also supports rich outputs such as charts. Type in the following code in the Code cell.

# Conceptualizing Python in Google COLAB

---

```
import numpy as np
from matplotlib import pyplot as plt
y = np.random.randn(100)
x = [x for x in range(len(y))]
plt.plot(x, y, '-')
plt.fill_between(x, y, 200, where = (y > 195), facecolor='g', alpha=0.6)
plt.title("Sample Plot")
plt.show()
```

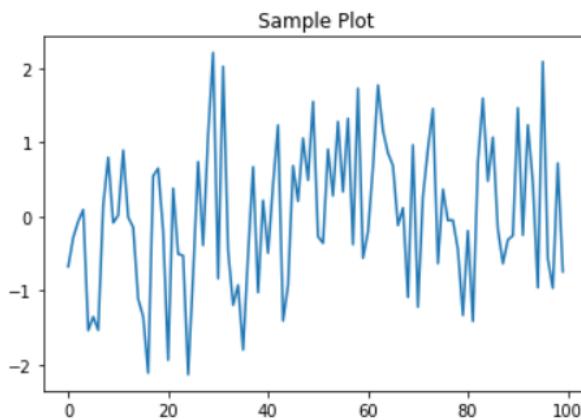


```
import numpy as np
from matplotlib import pyplot as plt

y = np.random.randn(100)
x = [x for x in range(len(y))]

plt.plot(x, y, '-')
plt.fill_between(x, y, 200, where = (y > 195), facecolor='g', alpha=0.6)

plt.title("Sample Plot")
plt.show()
```



## Code Editing Help

The present day developers rely heavily on context-sensitive help to the language and library syntaxes. That is why the IDEs are widely used. The Colab notebook editor provides this facility.

# Conceptualizing Python in Google COLAB

---

**Step 1** – Open a new notebook and type in the following code in the Code cell –

```
import numpy
```

**Step 2** – Run the code by clicking on the Run icon in the left panel of the Code cell. Add another Code cell and type in the following code –

```
numpy.
```

The screenshot shows a Jupyter Notebook cell with the following code:

```
[39] import numpy
```

Below the code, the user has typed "numpy." followed by a TAB key. A context help dropdown menu is displayed, listing various methods and constants of the NumPy module, such as abs, absolute, add, add\_docstring, add\_newdoc, add\_newdoc\_ufunc, alen, all, allclose, ALLOW\_THREADS, alltrue, and amax. The method "abs" is currently highlighted in blue, and its documentation is visible in the top right corner of the dropdown.

Note: If you do not run the cell containing import, the context help is not displayed.

## Function Documentation

Colab gives you the documentation on any **function** or **class** as a context-sensitive help.

Type the following code in your code window –

```
import numpy as np
```

*in one cell, run it and enter the following code in another cell.*

```
np.add(
```

Now, hit **TAB** and you will see the documentation on **cos** in the popup window as shown in the screenshot here. Note that you need to type in **open parenthesis** before hitting TAB.

# Conceptualizing Python in Google COLAB

```
[2] import numpy as np

np.add()
np.add(*args, **kwargs)

add(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Add arguments element-wise.

Parameters

x1, x2 : array_like
    The arrays to be added.
    If x1.shape != x2.shape , they must be broadcastable to a common
    shape (which becomes the shape of the output).
```

## Google Colab – Magics

Magics is a set of system commands that provide a mini extensive command language.

Magics are of two types –

- Line magics
- Cell magics

The line magics as the name indicates that it consists of a single line of command, while the cell magic covers the entire body of the code cell.

In case of line magics, the command is prepended with a *single % character* and in the case of cell magics, it is prepended with *two % characters (%)*.

Let us look into some examples of both to illustrate these.

## Line Magics

Type the following code in your code cell –

**%oldir**

The above command displays the content of current working directory.

# Conceptualizing Python in Google COLAB



```
%ldir
```

```
drwx----- 5 root 4096 May 28 16:25 drive/
drwxr-xr-x 1 root 4096 May  6 13:44 sample_data/
```

**%history**

This presents the complete history of commands that you have previously executed.



```
%history
```

```
import numpy as np
import numpy as np
%ldir
%history
```

## Cell Magics

Type in the following code in your code cell –

```
%%html
```

```
<marquee style='width: 50%; color: Green;'>Welcome to CSIBER!</marquee>
```

Now, if you run the code and you will see the scrolling welcome message on the screen as shown here –



```
%%html
```

```
<marquee style='width: 50%; color: Green;'>Welcome to CSIBER!</marquee>
```

Welcome to CSIBER!

To get a complete list of supported magics, execute the following command –

```
%lsmagic
```



```
Available line magics:
%alias %alias_magic %autocall %automagic %autosave %bookmark %cat %cd %clear %colors %config %connect_info %cp %debug

Available cell magics:
%%! %%HTML %%SVG %%bash %%bigquery %%capture %%debug %%file %%html %%javascript %%js %%latex %%perl %%prun %%pypy %

Automagic is ON, % prefix IS NOT needed for line magics.
```

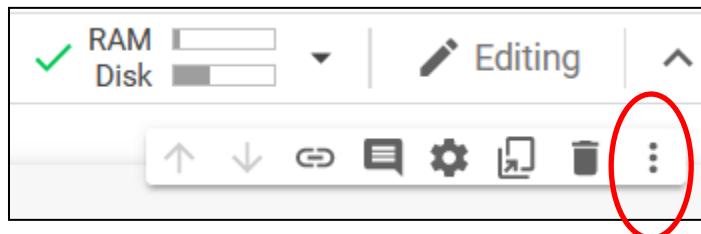
# Conceptualizing Python in Google COLAB

---

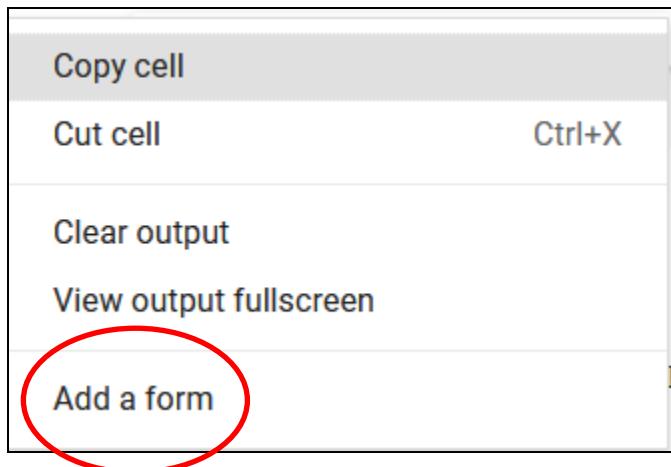
## Adding Form

For accepting input from user.

Click on vertically dotted '**Options**' menu in the right-top corner.



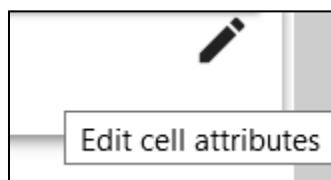
The following menu is displayed:



Now, select '**Add a form**' option. It will add the form to your Code cell with a Default title as seen in the screenshot here –



To change the title of the form, click on the '**Settings**' button (pencil icon on the right). It will pop up a settings screen as shown here:



'**Edit form attributes**' dialog appears as shown below:

# Conceptualizing Python in Google COLAB

## Edit form attributes

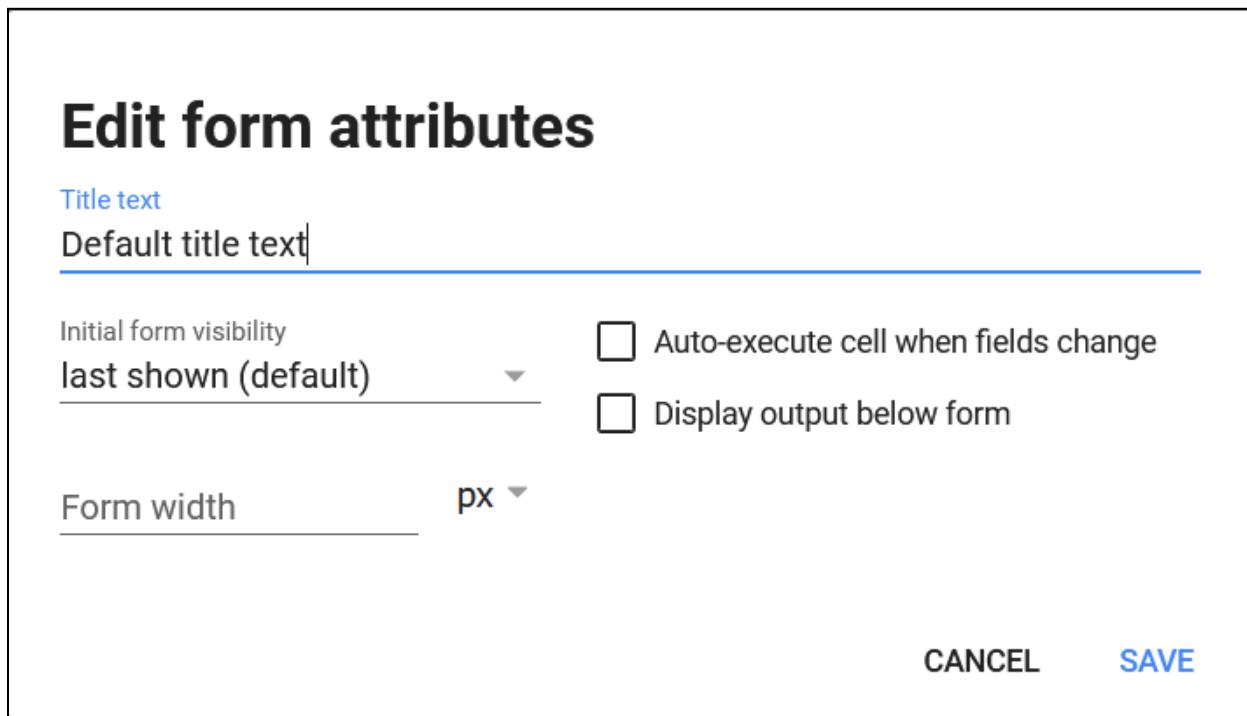
Title text  
Default title text

Initial form visibility  
last shown (default)

Form width px

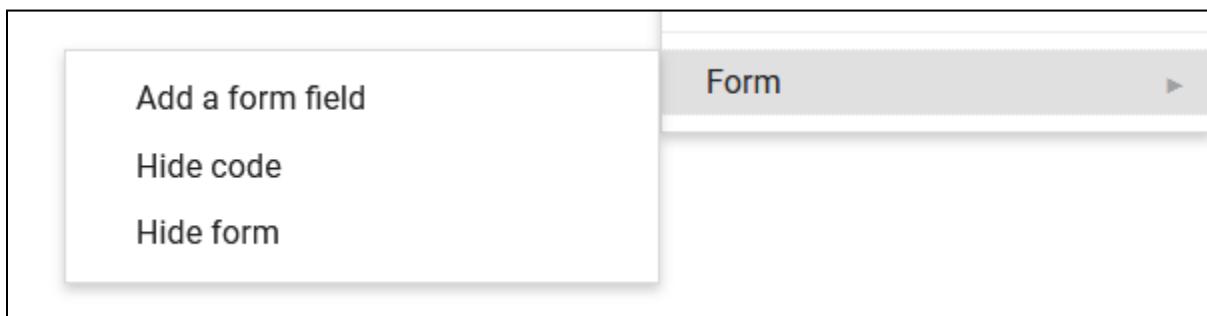
Auto-execute cell when fields change  
 Display output below form

CANCEL SAVE



## Adding Form Fields

To add a form field, click the '**Options**' menu in the Code cell, click on the '**Form**' to reveal the submenus. The screen will look as shown below –



Select '**Add a form field**' menu option. A dialog pops up as seen here –

Adding Form Fields –

# Conceptualizing Python in Google COLAB

**Add new form field**

Form field type <input type="text" value="input"/>	Variable type <input type="text" value="string"/>
Variable name <input type="text" value="variable_name"/>	

[CANCEL](#) [SAVE](#)

Leave the **Form field type** to ‘**input**’. Change the ‘**Variable name**’ to ‘**num**’ and set the ‘**Variable type**’ to ‘**number**’. Save the changes by clicking the **Save** button.

Your screen will now look like the following with the ‘**num**’ variable added into the code.

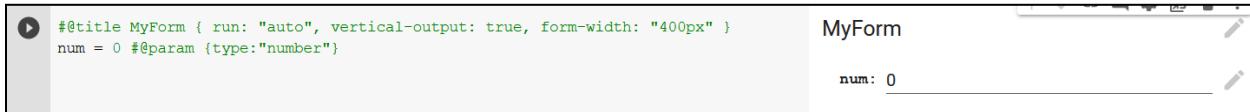
**Add new form field**

Form field type <input type="text" value="input"/>	Variable type <input type="text" value="number"/>
Variable name <input type="text" value="num"/>	

[CANCEL](#) [SAVE](#)

Your screen will now look like the following with the ‘**num**’ variable added into the code.

# Conceptualizing Python in Google COLAB



```
#@title MyForm { run: "auto", vertical-output: true, form-width: "400px" }
num = 0 #@param {type:"number"}
```

MyForm

num: 0

## Testing Form

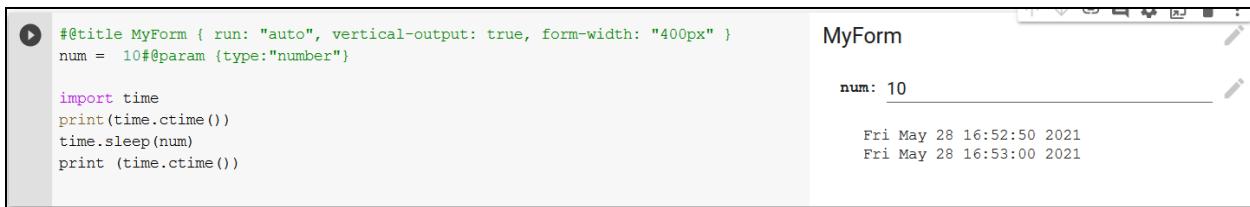
Add a new Code cell underneath the form cell. Use the code given below –

```
import time

print(time.ctime())

time.sleep(num)

print (time.ctime())
```



```
#@title MyForm { run: "auto", vertical-output: true, form-width: "400px" }
num = 10 #@param {type:"number"}

import time
print(time.ctime())
time.sleep(num)
print (time.ctime())
```

MyForm

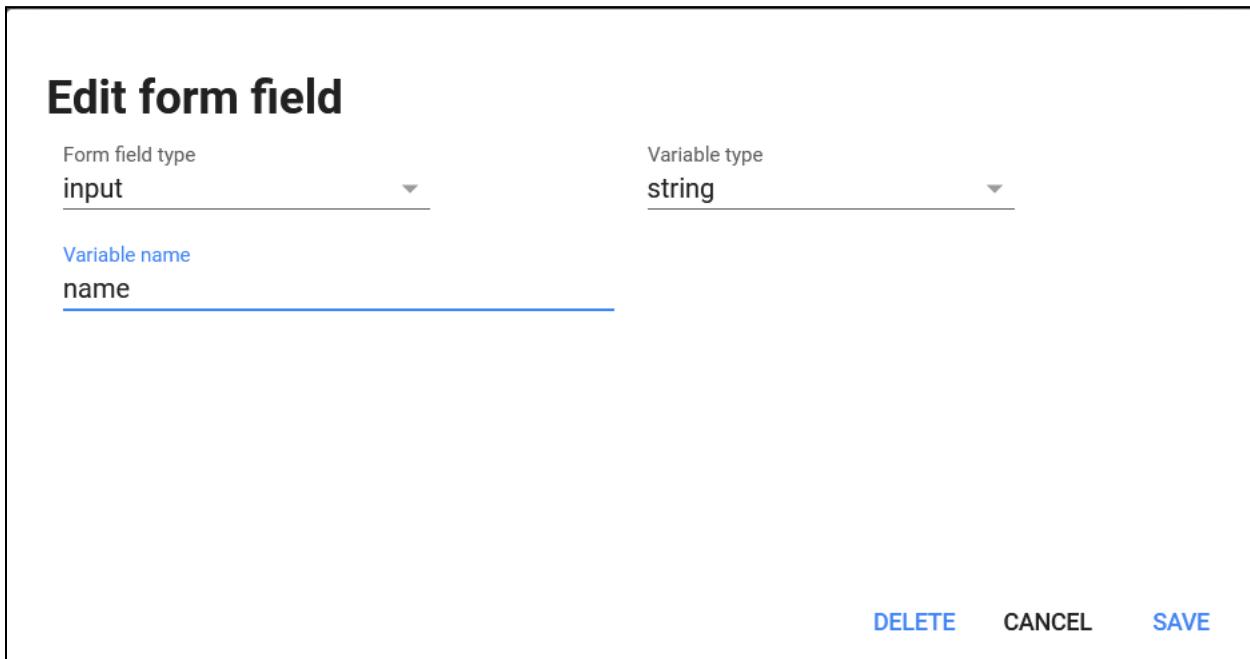
num: 10

Fri May 28 16:52:50 2021  
Fri May 28 16:53:00 2021

The output is displayed below the form control.

Edit the form as shown below:

Change the variable type to ‘string’ and variable name to ‘name’.



**Edit form field**

Form field type	Variable type
input	string
Variable name	
name	

DELETE CANCEL SAVE

# Conceptualizing Python in Google COLAB

---

Edit the form code as shown:

```
print("Hello",name)
```

Following output is displayed:

The screenshot shows a Jupyter notebook cell containing the following code:

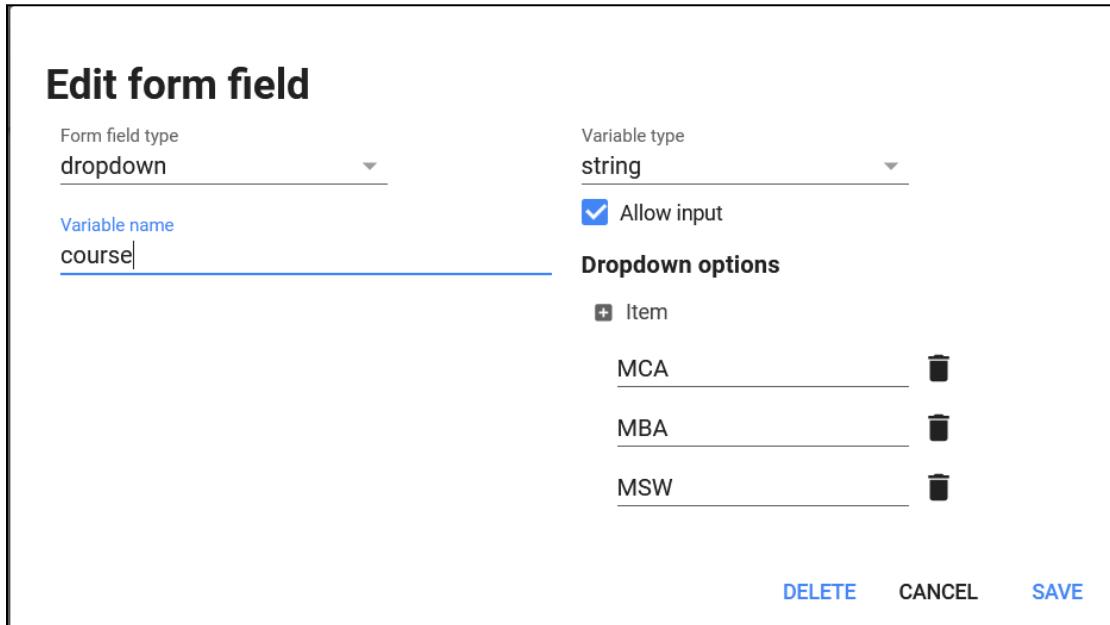
```
#@title MyForm { run: "auto", vertical-output: true, form-width: "400px" }
name = "MCA" #@param {type:"string"}

print("Hello",name)
```

To the right of the code, there is a preview window titled "MyForm". It shows a text input field with the value "MCA" and a text output below it saying "Hello MCA".

## Using Dropdown List

- Click on '**Form Field Type**' dropdown and select '**dropdown**' from the list displayed.
- Enter the variable type as '**string**' and variable name as '**course**'



- If the checkbox '**Allow Input**' is selected, then the user will be able to enter the new item, if the corresponding item does not exist in the list.

## Adding Item to the Dropdown List

- For adding new item to the list, click in '+' icon next to item and enter the name of the item in the textbox that appears.

# Conceptualizing Python in Google COLAB

**Edit form field**

Form field type: dropdown

Variable type: string

Variable name: course

Allow input

**Dropdown options**

+ Item  
MCA  
MBA  
MSW

**Buttons:** DELETE CANCEL SAVE

Edit the form code as shown below:

```
print("You Selected - ",course)
```

Following output is displayed:

The screenshot shows a Jupyter notebook cell containing Python code. The code defines a dropdown menu with options MCA, MBA, and MSW, and prints the selected value. To the right, the interface shows the dropdown menu open with 'MSW' selected, and the output cell below it displays the text 'You Selected - MSW'.

```
#@title MyForm { run: "auto", vertical-output: true, form-width: "400px" }
course = "MSW" #@param ["MCA", "MBA", "MSW"] (allow-input: true)

print("You Selected - ",course)
```

Since the option ‘*Allow-execute cell when fields change*‘ is checked when new item is selected from the drop-down list, the code in the cell is auto executed and the output is refreshed.

This screenshot shows the same setup as the previous one, but after changing the selection in the dropdown. The dropdown now shows 'MCA' instead of 'MSW', and the output cell has been updated to reflect this change, displaying 'You Selected - MCA'.

```
#@title MyForm { run: "auto", vertical-output: true, form-width: "400px" }
course = "MSW" #@param ["MCA", "MBA", "MSW"] (allow-input: true)

print("You Selected - ",course)
```

# Conceptualizing Python in Google COLAB

## Edit form attributes

Title text  
MyForm

Initial form visibility  
last shown (default)

Form width  
400 px

Auto-execute cell when fields change  
 Display output below form

CANCEL SAVE

## Date Input

Colab Form allows you to accept dates in your code with validations.

## Add new form field

Form field type  
input

Variable type  
date

Variable name  
start\_date

CANCEL SAVE

Colab Form allows you to accept dates in your code with validations. Use the following code to input date in your code.

# Conceptualizing Python in Google COLAB

---

```
#@title Default title text  
  
start_date = "2021-05-29" #@param {type:"date"}  
  
print(start_date)
```

The Form screen looks like the following.



## Installing ML Libraries

Colab supports most of machine learning libraries available in the market. Let us take a quick overview of how to install these libraries in your Colab notebook.

To install a library, you can use either of these options –

***!pip install***

or

***!apt-get install***

### Keras

Keras, written in Python, runs on top of TensorFlow, CNTK, or Theano. It enables easy and fast prototyping of neural network applications. It supports both convolutional networks (CNN) and recurrent networks, and also their combinations. It seamlessly supports GPU.

To install Keras, use the following command –

***!pip install -q keras***

### PyTorch

PyTorch is ideal for developing deep learning applications. It is an optimized tensor library and is GPU enabled. To install PyTorch, use the following command –

***!pip3 install torch torchvision***

# Conceptualizing Python in Google COLAB

---

## MxNet

Apache MxNet is another flexible and efficient library for deep learning. To install MxNet execute the following commands –

```
!apt install libnvrtc8.0
```

```
!pip install mxnet-cu80
```

## OpenCV

OpenCV is an open source computer vision library for developing machine learning applications. It has more than 2500 optimized algorithms which support several applications such as recognizing faces, identifying objects, tracking moving objects, stitching images, and so on. Giants like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota use this library. This is highly suited for developing real-time vision applications. To install OpenCV use the following command –

```
!apt-get -qq install -y libsm6 libxext6 && pip install -q -U opencv-python
```

## XGBoost

XGBoost is a distributed gradient boosting library that runs on major distributed environments such as Hadoop. It is highly efficient, flexible and portable. It implements ML algorithms under the Gradient Boosting framework. To install XGBoost, use the following command –

```
!pip install -q xgboost==0.4a30
```

## GraphViz

Graphviz is an open source software for graph visualizations. It is used for visualization in networking, bioinformatics, database design, and for that matter in many domains where a visual interface of the data is desired. To install GraphViz, use the following command –

```
!apt-get -qq install -y graphviz && pip install -q pydot
```

Developing machine learning models requires high processing power. Colab provides free GPU for your notebooks.

# Conceptualizing Python in Google COLAB

---

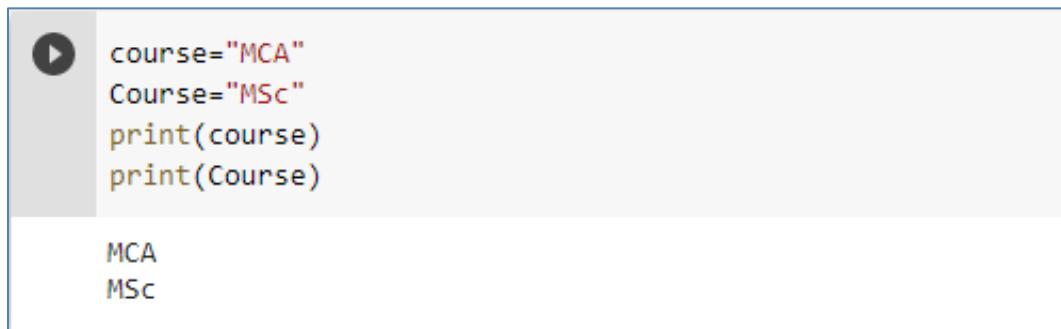
## Chapter 2

### Lab Assignments on Python Language Fundamentals

#### Level – Basic

##### Python Variables

Python variables are case-sensitive. In the following program, ‘*course*’ and ‘*Course*’ are treated as two distinct variables as demonstrated in the following program:

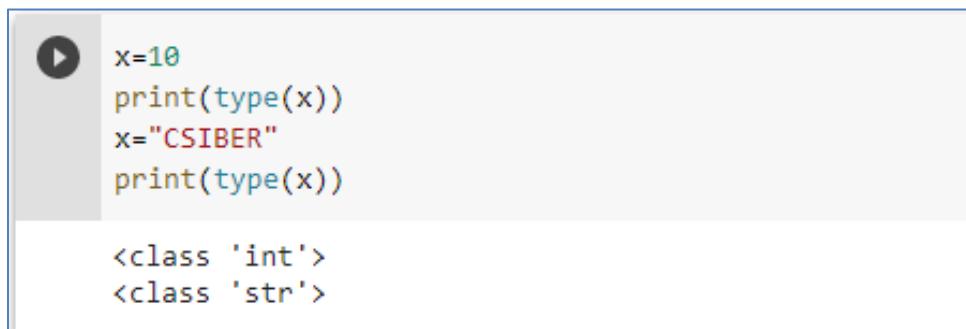


```
course="MCA"
Course="MSc"
print(course)
print(Course)

MCA
MSc
```

##### Checking Variable Type

Python is dynamically typed language. The type of the variable is decided at runtime based on the type of the value assigned to it. In the following program, the variable ‘*x*’ is initialized with the value 10, and hence ‘*x*’ is of type ‘*int*’. In the subsequent statement, the value of ‘*x*’ is changed to ‘*CSIBER*’ which changes the type of ‘*x*’ from ‘*int*’ to ‘*str*’ as demonstrated in the following program:



```
x=10
print(type(x))
x="CSIBER"
print(type(x))

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

# Conceptualizing Python in Google COLAB

---

## Size of Identifier in Python

The maximum possible length of an identifier is not defined in the python language. It can be of any number.

## Determining the size of Variable in Python

In Python ‘**int**’ is a full-fledged object which involves an extra overhead which stores reference count, object type among others. In the following figure, size of int class and instance of int class is displayed.

```
import sys
print(sys.getsizeof(int))
print(sys.getsizeof(int()))
```

400  
24

As shown in the following figure, the size of ‘**int**’ data type is augmented by 4 bytes for every  $2^{30}$  times increment in the value which accounts for large value that can be stored in ‘**int**’ data type.

```
import sys
print(sys.getsizeof(0))
print(sys.getsizeof(1))
print(sys.getsizeof(2**30-1))
print(sys.getsizeof(2**30))
print(sys.getsizeof(2**60-1))
print(sys.getsizeof(2**60))
```

24  
28  
28  
32  
32  
36

# Conceptualizing Python in Google COLAB

---

How bool value is stored in Python.

Similar to programming languages such as C and C++, Python employs the numeric values 0 and 1 for storing the boolean values '**False**' and '**True**', respectively.



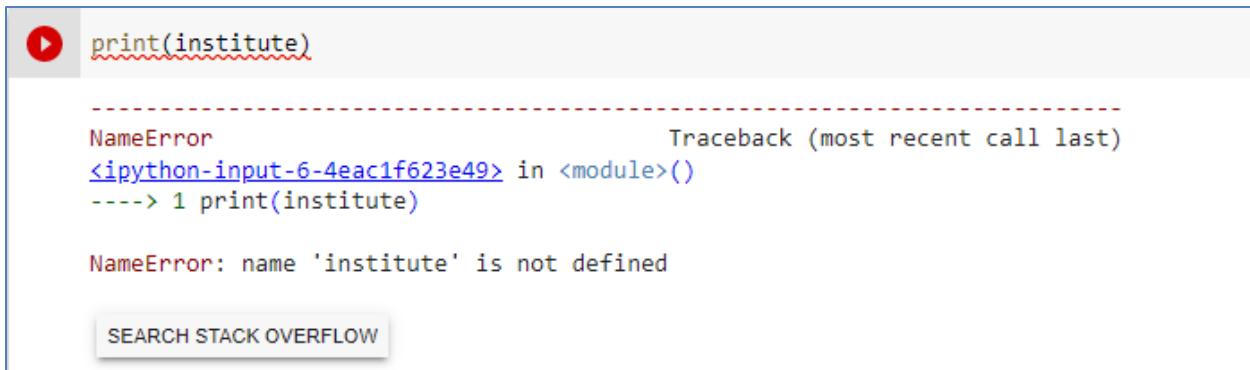
```
print(bool(0))
print(bool(1))

False
True
```

Hence '**True**' and '**False**' values are stored in Python using numeric 1 and 0, respectively.

## Variable Initialization Issues

Python does not automatically initialize the variables to the default values. Attempting to use the variable without initializing will generate '**NameError**' as demonstrated in the following program:



```
print(institute)

-----
NameError                                 Traceback (most recent call last)
<ipython-input-6-4eac1f623e49> in <module>()
      1 print(institute)

NameError: name 'institute' is not defined

SEARCH STACK OVERFLOW
```

## Data Type of Variables – type() Function

Python support `type()` function for determining runtime instance of a variable. `type()` function accepts a single parameter the variable whose type is to be determined as demonstrated in the following program:

# Conceptualizing Python in Google COLAB

---

```
x=10  
print(type(x))  
  
x=10.20  
print(type(x))  
  
x="CSIBER"  
print(type(x))
```

```
<class 'int'>  
<class 'float'>  
<class 'str'>
```

## Checking the Type of the Variable – `isinstance()`

Python supports a function `isinstance()` for checking the instance of the variable at runtime as demonstrated in the following figure. The `isinstance()` function accepts two parameters:

- first parameter is the name of the variable and
- second parameter is the class

```
i=10  
print(isinstance(i,int))  
print(isinstance(i,str))
```

```
True  
False
```

In the above program, ‘i’ is an instance of type ‘int’. Hence the first print statement displays ‘**True**’ and the second print statement displays ‘**False**’.

## Implicit Type Casting

Python automatically converts smaller data type to larger data type i.e. the result of adding ‘**int**’ and ‘**float**’ values is ‘**float**’ as demonstrated in the following program:

# Conceptualizing Python in Google COLAB

---

```
x=10
y=10.20
z=x+y
print(z)
print(type(z))

20.2
<class 'float'>
```

## Multiple Assignments in a Single Statement

Python enables initialization of multiple values in a single statement employing ‘=’ operator. The ‘=’ operator is right associative i.e. it evaluates the expression from right to left. In the following program, the constant ‘1’ is assigned to a variable ‘c’ which is then assigned to ‘a’ and ‘b’ subsequently.

```
a=b=c=1
print(a)
print(b)
print(c)

→ 1
1
1
```

In the following program, the variables ‘a’, ‘b’ and ‘c’ are initialized to a string constant ‘CSIBER’ in a single statement.

```
a=b=c="CSIBER"
print(a)
print(b)
print(c)

CSIBER
CSIBER
CSIBER
```

# Conceptualizing Python in Google COLAB

---

## Deleting a variable in Python (NameError)

A reference to the variable can be deleted using ‘**del**’ operator. The syntax for using ‘**del**’ operator is shown below:

```
del <var_1>[, <var_2>, ..., <var_N>]
```

Attempting to access a deleted variable later in the program generates ‘NameError’ as shown in the following program:

The screenshot shows a Jupyter Notebook cell with the following code:

```
a=b=c="CSIBER"
del a,b
print(c)
print(a)
```

When run, it outputs:

```
CSIBER
-----
NameError                                 Traceback (most recent call last)
<ipython-input-5-8e185d6c1196> in <module>()
      2 del a,b
      3 print(c)
----> 4 print(a)

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

Below the cell, there is a button labeled "SEARCH STACK OVERFLOW".

In the above program, the variables ‘**a**’ and ‘**b**’ are deleted. An attempt to access the deleted variable will generate ‘**NameError**’ as shown above:

## Using Escape Characters in Strings

The escape character is used for performing a specific action on a string. For example, ‘**\n**’ characters are used for appending new line character at the end of a string. ‘**\t**’ is used for using tab space between the words as demonstrated in the following programs:

The screenshot shows a Jupyter Notebook cell with the following code:

```
print("This is first line\nThis is second line")
```

When run, it outputs:

```
This is first line
This is second line
```

# Conceptualizing Python in Google COLAB

---

```
▶ print("Rollno\tName\tDivision")
```

```
Rollno  Name  Division
```

## Raw String

Raw string can be displayed by prepending the character ‘r’ to the string. Raw string displays the escape characters without any interpretation as shown in the following programs:

```
▶ print(r"This is first line\nThis is second line")
```

```
This is first line\nThis is second line
```

```
▶ print(r"Rollno\tName\tDivision")
```

```
Rollno\tName\tDivision
```

## String Repetition Operator

\* operator is used for repeating the string the specified no. of times. In the following example, the string ‘CSIBER’ is repeated 5 times.

```
▶ print(5*"CSIBER ")
```

```
CSIBER CSIBER CSIBER CSIBER CSIBER
```

## String Indexing

Python enables traversing the string in both directions using positive and negative index, respectively. For the string of length ‘ $n$ ’ the indexing in both the directions are depicted in the following table:

# Conceptualizing Python in Google COLAB

<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	..	<i>n</i>
<i>-n</i>	<i>1-n</i>	<i>2-n</i>	<i>3-n</i>	<i>4-n</i>	<i>5-n</i>	<i>6-n</i>	..	<i>-1</i>



```
str1="SIBER"
for i in range(5):
    print(i,str1[i])
print("\n")
for i in range(-5,0,1):
    print(i,str1[i])
```

```
0 S
1 I
2 B
3 E
4 R
```

```
-5 S
-4 I
-3 B
-2 E
-1 R
```

In the above example, the different characters of the string ‘CSIBER’ are traversed in both the directions using the forward and backward indexing mechanisms.

## Bounds Checking for Strings

Python performs strict bounds checking on arrays and strings. Attempting to access an element beyond limits generates an ‘**IndexError**’ as shown in the following program:

# Conceptualizing Python in Google COLAB

```
▶ str1="SIBER"  
print(str1[5])  
  
-----  
IndexError Traceback (most recent call last)  
<ipython-input-27-b1e0a12b045f> in <module>()  
      1 str1="SIBER"  
----> 2 print(str1[5])  
  
IndexError: string index out of range  
  
SEARCH STACK OVERFLOW
```

## A Special Data Type – None

‘None’ is used for defining a null value. In Python a variable cannot be used without initializing it. Using an uninitialized variable generates ‘NameError’ as shown in the following figure:

```
▶ print(x)  
  
-----  
NameError Traceback (most recent call last)  
<ipython-input-5-fc17d851ef81> in <module>()  
----> 1 print(x)  
  
NameError: name 'x' is not defined  
  
SEARCH STACK OVERFLOW
```

If the value of the variable is not known at the time of declaration, then it can be initialized with ‘None’ keyword. ‘None’ is not same as zero, False, or empty string. ‘None’ is an instance of ‘NoneType’ class as shown in the following program:

```
▶ print(type(None))  
  
<class 'NoneType'>
```

# Conceptualizing Python in Google COLAB

---

```
x = None
if x is None:
    print("x is not initialized")
else:
    print("x is initialized and has value ",x)

x=10
if x is None:
    print("x is not initialized")
else:
    print("x is initialized and has value ",x)

x is not initialized
x is initialized and has value  10
```

Note: If a function does not return any value, then it returns ‘None’ . In the following program, f is a function which does not return any value. The last statement in the program, prints the value returned by the function f() which is ‘None’ .

```
def f():
    print("No return value")
print(f())

No return value
None
```

```
list1=[1,2,2,3,4,4,5]
set1=set(list1)
print(list1)
print(set1)

[1, 2, 2, 3, 4, 4, 5]
{1, 2, 3, 4, 5}
```

## List in Python

A list is data structure which contains comma separated heterogeneous elements enclosed within a pair of square brackets ([]).

# Conceptualizing Python in Google COLAB

---

## Features of List:

- List is mutable.
- List maintains insertion order.
- List is iterable.
- List is ordered.

## Operations on List

### Slicing Operator

A slicing operator is employed for selecting multiple contiguous elements from the list. For retrieving the elements from the index position ‘i’ to ‘j’ in a list ‘list1’ use the following syntax:

*list1[ i : j+1]*



```
l=[10, 20.30, "Python", 'c', 23]
l1=['Msc','CSIBER']
print(l[0])
print(l[1:3])
print(l[:2])
print(l[2:])
print(l*2)
print(l+l1)

10
[20.3, 'Python']
[10, 20.3]
['Python', 'c', 23]
[10, 20.3, 'Python', 'c', 23, 10, 20.3, 'Python', 'c', 23]
[10, 20.3, 'Python', 'c', 23, 'Msc', 'CSIBER']
```

### Negative Indexing

For retrieving the m elements from the right in a list ‘list1’ use the following syntax:

*list1[ -m-1 : -1]*

# Conceptualizing Python in Google COLAB

## Example:

For retrieving last two elements from the 'l' shown below,

`m=2`

$-m-1 = -2-1 = -3$

`list1[-3, -1]`

```
▶ l=[10, 20.30, "Python", 'c', 23]
  print(l[-3:-1])

  ['Python', 'c']
```

For retrieving the last four elements from the list use the following statement:

`l[-5 : -1]`

```
▶ l=[10, 20.30, "Python", 'c', 23]
  print(l[-5:-1])

  [10, 20.3, 'Python', 'c']
```

```
▶ l=[10, 20.30, "Python", 'c', 23]
  print(l[-1])
  print(l[-3:-1])
  print(l[-2:])
  print(l[:-3])

  23
  ['Python', 'c']
  ['c', 23]
  [10, 20.3]
```

## Retrieving the elements Using Positive and Negative Indexing

The positive and negative index positions for a list of length 10 is shown in the following figure:

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

# Conceptualizing Python in Google COLAB

---

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

The relationship between the positive and negative indexing is given by:

$$l[n] = l[n-s]$$

where s is the size of the list.

For retrieving the elements from the index position ‘i’ to ‘j’ in a list ‘list1’ using positive indexing, use the following syntax:

*list1[i : j+1]*

For retrieving the elements from the index position ‘i’ to ‘j’ in a list ‘list1’ using negative indexing, use the following syntax:

*list1[i - s : j + 1 - s]*

where, s is the size of the list.

**Example:**

For retrieving the elements from the index positions 1 to 3 using the positive indexing use the following statement:

`l[1:4]`

```
▶ 1=[10, 20.30, "Python", 'c', 23]
   print(l[1:4])

[20.3, 'Python', 'c']
```

For retrieving the elements from the index positions 1 to 3 using the negative indexing use the following statement:

`l[-4:-1]`

```
▶ 1=[10, 20.30, "Python", 'c', 23]
   print(l[-4:-1])

[20.3, 'Python', 'c']
```

# Conceptualizing Python in Google COLAB

---

## String Slicing Operator

The slicing operator can also be employed for stripping substrings of a given string employing the same syntax depicted above.

```
▶ str="Hello World"
  print(str[0])
  print(str[2:5])
  print(str[:4])
  print(str[4:])
  print(str*4)
  print(str+ " and Planet")
```

```
H
llo
Hell
o World
Hello WorldHello WorldHello WorldHello World
Hello World and Planet
```

## List is Mutable

List is mutable implies the elements of the list can be modified after creating the list.

```
▶ l=[10, 20.30, "Python", 'c', 23]
  print(l)
  l[2]="Java"
  print(l)
```

```
[10, 20.3, 'Python', 'c', 23]
[10, 20.3, 'Java', 'c', 23]
```

## List is Iterable

The different elements of the list can be traversed using for loop as shown in the following program:

# Conceptualizing Python in Google COLAB

---

```
▶ list1=[1,2,3,4,5]
  for i in list1:
    print(i)
```

```
1
2
3
4
5
```

## List is Ordered

Two lists are considered to be equal if and only if they contain same elements also in the same order. In the following program, list1, list2 and list3 contain the same elements. However, list1 and list3 contain the elements in the same order while list1 and list2 contain the same elements in different order. Hence list1 and list3 are considered to be equal while list1 and list2 are not equal as demonstrated in the following example:

```
▶ list1=[1,2,3,4,5]
  list2=[5,4,3,2,1]
  list3=[1,2,3,4,5]
  if (list1==list2):
    print("list1 and list2 are equal")
  else:
    print("list1 and list2 are not equal")

  if (list1==list3):
    print("list1 and list3 are equal")
  else:
    print("list1 and list3 are not equal")
```

↳ list1 and list2 are not equal  
list1 and list3 are equal

## List Concatenation

The two lists can be concatenated using ‘+’ operator as demonstrated in the following program:

# Conceptualizing Python in Google COLAB



```
list1=[1,2,3,4,5]
list2=[6,7,8,9,10]
list3=list1+list2
print(list1)
print(list2)
print(list3)
```

```
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

**Rule 1:** A list and a tuple cannot be concatenated.

Attempting to concatenate a tuple with the list generates '**TypeError**'.



```
list1=[1,2,3,4,5]
tuple1=(6,7,8,9,10)
list3=list1+tuple1
print(list1)
print(tuple1)
print(list3)
```



```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-17-6888cb91ade7> in <module>()  
      1 list1=[1,2,3,4,5]  
      2 tuple1=(6,7,8,9,10)  
----> 3 list3=list1+tuple1  
      4 print(list1)  
      5 print(tuple1)  
  
TypeError: can only concatenate list (not "tuple") to list
```

[SEARCH STACK OVERFLOW](#)

**Rule 2:** A single element cannot be added to a list, since '**int**' type is not iterable. Doing so will generate '**TypeError**' as shown in the following figure:

# Conceptualizing Python in Google COLAB



```
list1=[1,2,3,4,5]
list2=list1+10
print(list1)
print(list2)
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-18-213ca2f7495d> in <module>()  
      1 list1=[1,2,3,4,5]  
----> 2 list2=list1+10  
      3 print(list1)  
      4 print(list2)  
  
TypeError: can only concatenate list (not "int") to list
```

SEARCH STACK OVERFLOW

However, the following program compiles successfully.



```
list1=[1,2,3,4,5]
list2=list1+[10]
print(list1)
print(list2)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 10]
```

## Checking the Existence of an Element in a List

The membership operator ‘**in**’ can be used for testing the existence of an element in a list as demonstrated in the following program:

In the following program, the elective subject is accepted from the end user and is then tested for the availability in ‘**electives**’ list.

# Conceptualizing Python in Google COLAB

```
electives=["Machine Learning","Big Data Analytics","Kotlin","MEAN Stack"]
elective=input("Choose Elective")
if (elective in electives):
    print(elective, " is offered as an elective subject")
else:
    print(elective, " is not offered as an elective subject")
```

```
Choose ElectiveMEAN Stack
MEAN Stack  is offered as an elective subject
```

```
electives=["Machine Learning","Big Data Analytics","Kotlin","MEAN Stack"]
elective=input("Choose Elective")
if (elective in electives):
    print(elective, " is offered as an elective subject")
else:
    print(elective, " is not offered as an elective subject")
```

```
Choose ElectiveBlockchain
Blockchain  is not offered as an elective subject
```

## Determining the Size of the List – len() Function

*len()* function can be used for determining the size of the list which returns the no. of elements available in the list.

```
electives=["Machine Learning","Big Data Analytics","Kotlin","MEAN Stack"]
print("No of electives offered is : ",len(electives))

No of electives offered is : 4
```

## Appending an Element to a List –append() Method

*append()* method can be used for appending a single element to the list as shown in the following program:

# Conceptualizing Python in Google COLAB

```
▶ list1=[1,2,3,4,5]
    print("Before Appending : ",list1)
    list1.append(6)
    print("After Appending  : ",list1)

Before Appending :  [1, 2, 3, 4, 5]
After Appending  :  [1, 2, 3, 4, 5, 6]
```

## extend() Method

For appending more than one element n a single go, instead use extend() method which accepts a list as its argument, iterates through the list and appends each element to the list on which extend() is invoked.

```
▶ list1=[1,2,3,4,5]
    print("Before Appending : ",list1)
    list1.extend([6,7,8,9,10])
    print("After Appending  : ",list1)

↳ Before Appending :  [1, 2, 3, 4, 5]
    After Appending  :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Invoking *extend()* method on a list by passing a parameter which is not iterable generates ‘*TypeError*’ as shown in the following program:

```
▶ list1=[1,2,3,4,5]
    print("Before Appending : ",list1)
    list1.extend(6)
    print("After Appending  : ",list1)

Before Appending :  [1, 2, 3, 4, 5]
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-6-e8c375987729> in <module>()
      1 list1=[1,2,3,4,5]
      2 print("Before Appending : ",list1)
----> 3 list1.extend(6)
      4 print("After Appending  : ",list1)

TypeError: 'int' object is not iterable
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

---

## Difference Between append() and extend() Methods

The **append()** method is used for appending a single element to the list and accepts a single parameter, the element to be appended to the list. On the contrary, the **extend()** method accepts a list as its only argument and appends all the elements to the list by iterating through it.

When an **append()** method is invoked on a list by passing another list as a parameter, the list passed in the parameter is appended as an element to the list creating a nested list as demonstrated in the following program:

```
▶ electives=["Machine Learning","Big Data Analytics","Kotlin","MEAN Stack"]
  electives.append(["Cyber Security"])
  print(electives)

['Machine Learning', 'Big Data Analytics', 'Kotlin', 'MEAN Stack', ['Cyber Security']]
```

On modifying the above program by passing a string to append() method, the string is appended to the list as shown below:

```
▶ electives=["Machine Learning","Big Data Analytics","Kotlin","MEAN Stack"]
  electives.append("Cyber Security")
  print(electives)

['Machine Learning', 'Big Data Analytics', 'Kotlin', 'MEAN Stack', 'Cyber Security']
```

On passing a string argument to extend() method, the individual characters of a string are appended to the list as shown in the following figure:

```
▶ electives=["Machine Learning","Big Data Analytics","Kotlin","MEAN Stack"]
  electives.extend("Cyber Security")
  print(electives)

['Machine Learning', 'Big Data Analytics', 'Kotlin', 'MEAN Stack', 'C', 'y', 'b', 'e', 'r', ' ', 'S', 'e', 'c', 'u', 'r', 'i', 't', 'y']
```

To append a string to the list, modify the above program as shown below.

```
▶ electives=["Machine Learning","Big Data Analytics","Kotlin","MEAN Stack"]
  electives.extend(["Cyber Security"])
  print(electives)

['Machine Learning', 'Big Data Analytics', 'Kotlin', 'MEAN Stack', 'Cyber Security']
```

# Conceptualizing Python in Google COLAB

```
▶ numbers=[10, 20, 30, 40]
numbers.extend(50)
print(numbers)

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-32-dec628aaaf039> in <module>()
      1 numbers=[10, 20, 30, 40]
----> 2 numbers.extend(50)
      3 print(numbers)

TypeError: 'int' object is not iterable
```

SEARCH STACK OVERFLOW

## Cloning a List

Assigning a list to another list simply copies a reference and will not generate a new list. A single list is referenced by two distinct references. The list can be modified employing any of the two references. In the following program, the references ‘**numbers**’ and ‘**numbers1**’ refer to the same list [10, 20, 30, 40]. The elements at the index position 0 and 3 are modified employing reference ‘**numbers1**’.

```
▶ numbers=[10, 20, 30, 40]
numbers1=numbers
print(numbers)
print(numbers1)
numbers1[0]=100
numbers1[3]=400
print("After modifying numbers1 list : ")
print(numbers)
print(numbers1)

[10, 20, 30, 40]
[10, 20, 30, 40]
After modifying numbers1 list :
[100, 20, 30, 400]
[100, 20, 30, 400]
```

## Object Cloning – **copy()** Method

Python supports **copy()** method for cloning the object by duplicating the contents. **copy()** method creates an exact copy of the object by duplicating the contents and returns a reference to the newly created object. This enables two copies to be manipulated independently of each other. In the following program, the variables ‘**numbers**’ and ‘**numbers1**’ refer to two distinct objects. Hence

# Conceptualizing Python in Google COLAB

modifying the object referenced by ‘**numbers1**’ does not modify the object referenced by ‘**numbers**’.

```
▶ numbers=[10, 20, 30, 40]
numbers1=numbers.copy()
print(numbers)
print(numbers1)
numbers1[0]=100
numbers1[3]=400
print("After modifying numbers1 list : ")
print(numbers)
print(numbers1)

[10, 20, 30, 40]
[10, 20, 30, 40]
After modifying numbers1 list :
[10, 20, 30, 40]
[100, 20, 30, 400]
```

## Testing Object Equivalence Using ‘is’ Operator

‘**is**’ operator accepts two operands and returns ‘**True**’ if the two references point to the same object, otherwise returns ‘**False**’.

```
▶ list1=[1,2,3,4,5]
list2=list1
list3=[1,2,3,4,5]
print(list1)
print(list1 is list2)
print(list1 is list3)

⇨ [1, 2, 3, 4, 5]
True
False
```

In the above program, ‘**list1**’ and ‘**list2**’ are two different references referencing the same List object. On the contrary, ‘**list2**’ and ‘**list3**’ are two different references referencing the distinct List objects.

# Conceptualizing Python in Google COLAB

---

## Shallow Copying Vs. Deep Copying

### Shallow Copying

Shallow copying only copies the outer structure i.e. the elements of the outer list and references to inner lists, if any. As a result, on shallow copying a nested list, the elements of the outer list can be manipulated independently of each other while the inner lists are shared. In the following program, ‘numbers’ is a nested list which is cloned using *copy()* method in a new reference ‘**numbers1**’. The inner list is manipulated using a reference ‘**numbers1**’. Since the copying is shallow, ‘**numbers**’ list reflects the changes carried out by ‘**numbers1**’ as shown in the following program:

```
▶ numbers=[10, 20, 30, 40, [50, 60, 70]]
  numbers1=numbers.copy()
  print(numbers)
  print(numbers1)
  numbers1[4][0]=500
  numbers1[4][1]=600
  numbers1[4][2]=700
  print("After modifying numbers1 list : ")
  print(numbers)
  print(numbers1)

[10, 20, 30, 40, [50, 60, 70]]
[10, 20, 30, 40, [50, 60, 70]]
After modifying numbers1 list :
[10, 20, 30, 40, [500, 600, 700]]
[10, 20, 30, 40, [500, 600, 700]]
```

### Deep Copying

For deep copying a nested list Python supports a *deepcopy()* method which creates an exact copy of complete nested structure as demonstrated in the following program. In the following example, the above program is re-written for deep copying a list. Hence manipulating the inner list using the reference ‘**numbers1**’ does not modify the list referenced by ‘**numbers**’.

# Conceptualizing Python in Google COLAB

```
import copy  
numbers=[10, 20, 30, 40, [50, 60, 70]]  
numbers1=copy.deepcopy(numbers)  
print(numbers)  
print(numbers1)  
numbers1[4][0]=500  
numbers1[4][1]=600  
numbers1[4][2]=700  
print("After modifying numbers1 list : ")  
print(numbers)  
print(numbers1)
```

```
[10, 20, 30, 40, [50, 60, 70]]  
[10, 20, 30, 40, [50, 60, 70]]  
After modifying numbers1 list :  
[10, 20, 30, 40, [50, 60, 70]]  
[10, 20, 30, 40, [500, 600, 700]]
```

## Sorting a List

Python supports `sort()` method for sorting a list. The `sort()` method returns '*None*'. The list on which the method is invoked is sorted and no new list is created as shown in the following program.

```
numbers=[1, 6, 344, 22, 17]  
sorted_numbers=numbers.sort()  
print(type(sorted_numbers))  
  
<class 'NoneType'>
```

# Conceptualizing Python in Google COLAB

---

```
▶ numbers=[1, 6, 344, 22, 17]
print(id(numbers))
sorted_numbers=numbers.sort()
print(type(sorted_numbers))
print(id(numbers))
print(numbers)
print(sorted_numbers)
```

```
140356966316944
<class 'NoneType'>
140356966316944
[1, 6, 17, 22, 344]
None
```

As seen by the output generated above, the '*id*' of list '*numbers*' before and after sorting is same. Hence no new list is created. The original list is re-organized.

The following program sorts the list containing the elective subjects in ascending order using *sort()* method.

```
▶ import copy
electives=["PHP", "Python","Java","Kotlin","MEAN Stack"]
print("Before sorting : ")
print(electives)
electives.sort()
print("After sorting : ")
print(electives)

Before sorting :
['PHP', 'Python', 'Java', 'Kotlin', 'MEAN Stack']
After sorting :
['Java', 'Kotlin', 'MEAN Stack', 'PHP', 'Python']
```

## Sorting in Ascending and Descending Order – *sorted()* Method

Python supports a *sorted()* method for bi-directional sorting of elements. It can be used for sorting the elements either in ascending or descending order. Unlike *sort()* method of List class *sorted()* method accepts a list reference as a parameter, creates a new sorted list and returns the same.

# Conceptualizing Python in Google COLAB

```
▶ numbers=[1,6,344,22,17]
print(id(numbers))
sorted_numbers=sorted(numbers)
print(type(sorted_numbers))
print(id(sorted_numbers))
print(numbers)
print(sorted_numbers)
```

```
140356966293664
<class 'list'>
140356966858992
[1, 6, 344, 22, 17]
[1, 6, 17, 22, 344]
```

As seen by the output generated above, the '*id*' of original list and sorted list are different. Hence a new list is created after sorting. The following program sorts the list containing the elective subjects in both ascending and descending order using *sorted()* method.

```
▶ import copy
electives=["PHP", "Python","Java","Kotlin","MEAN Stack"]
print("Before sorting : ")
print(electives)
print()
asorted_electives=sorted(electives)
dsorted_electives=sorted(electives,reverse=True)
print("After sorting in Ascending Order : ")
print(asorted_electives)
print()
print("After sorting in Descending Order : ")
print(dsorted_electives)
```

```
Before sorting :
['PHP', 'Python', 'Java', 'Kotlin', 'MEAN Stack']

After sorting in Ascending Order :
['Java', 'Kotlin', 'MEAN Stack', 'PHP', 'Python']

After sorting in Descending Order :
['Python', 'PHP', 'MEAN Stack', 'Kotlin', 'Java']
```

# Conceptualizing Python in Google COLAB

---

## Deleting the Elements of a List – del Method

The del method can be used for one of the following:

- for deleting a single element from the list.
- for deleting range of elements
- for deleting the entire list along with the reference.

### Deleting a Single Element from the List Using del Method

**del** method is used for deleting a single or a range of elements from a list. An attempt to access the deleted element generates ‘**NameError**’.

```
▶ list1=[1,2,3,4,5]
      print("List Size Before Deletion :",len(list1))
      del list1[0]
      print(list1[0])
      print("List Size After Deletion  :",len(list1))

      List Size Before Deletion : 5
      2
      List Size After Deletion  : 4
```

### Deleting a Range of Elements from the List Using del Method

The syntax for deleting the elements in the index position n-m using positive indexing is

**del[n:m+1]**

The following program deletes the elements at the index position 2 to 4 using del method using positive indexing.

```
▶ list1=[1,2,3,4,5]
      print("List Size Before Deletion :",len(list1))
      print(list1)
      del list1[2:5]
      print("List Size After Deletion  :",len(list1))
      print(list1)

      List Size Before Deletion : 5
      [1, 2, 3, 4, 5]
      List Size After Deletion  : 2
      [1, 2]
```

The syntax for deleting the elements in the index position n-m using negative indexing is

# Conceptualizing Python in Google COLAB

---

***del[n-s:m-s+1]***

where s is the size of the list.

In the following example, the above program is re-written using negative indexing.

For the following list

[1, 2, 3, 4, 5]

s=5 n=2 and m=4

Hence  $n - s = 2 - 5 = -3$  and

$m-s+1 = 4-5+1 = 0$

```
▶ list1=[1,2,3,4,5]
print("List Size Before Deletion :",len(list1))
print(list1)
del list1[-3:]
print("List Size After Deletion  :",len(list1))
print(list1)

List Size Before Deletion : 5
[1, 2, 3, 4, 5]
List Size After Deletion  : 2
[1, 2]
```

## Deleting All Elements from the list with Reference Intact - clear() Method

clear() method is used for deleting all the elements from the list while keeping the reference intact.

The list reference is not deleted but points to an empty list on invoking clear() method as demonstrated in the following program:

# Conceptualizing Python in Google COLAB

```
list1=[1,2,3,4,5]
print("List Size Before Deletion :",len(list1))
print(list1)
del list1[:]
print("List Size After Deletion  :",len(list1))
print(list1)

List Size Before Deletion : 5
[1, 2, 3, 4, 5]
List Size After Deletion  : 0
[]
```

## Deleting All Elements from the list Along with Reference

del method can be used for deleting the entire list along with the reference. An attempt to access the deleted list generates '*NameError*' as demonstrated in the following program:

```
list1=[1,2,3,4,5]
print("List Size Before Deletion :",len(list1))
print(list1)
del list1
print("List Size After Deletion  :",len(list1))
print(list1)

List Size Before Deletion : 5
[1, 2, 3, 4, 5]
-----
NameError                                 Traceback (most recent call last)
<ipython-input-18-60733a935b78> in <module>()
      3 print(list1)
      4 del list1
----> 5 print("List Size After Deletion  :",len(list1))
      6 print(list1)

NameError: name 'list1' is not defined
```

SEARCH STACK OVERFLOW

## Deleting the Element of a List Using pop() Method

pop() method can be used for deleting an element at the highest index position.

```
list1=[1,2,3,4,5]
print("List Before pop operation : ", list1)
list1.pop()
print("List After pop operation : ",list1)

List Before pop operation :  [1, 2, 3, 4, 5]
List After pop operation :  [1, 2, 3, 4]
```

# Conceptualizing Python in Google COLAB

pop() method also accepts an index of the element to be deleted which is a default parameter. When invoked without any parameter pop() method deletes an element at highest index position.

```
▶ list1=[1,2,3,4,5]
      print("List Before pop operation : ", list1)
      list1.pop(0)
      print("List After pop operation : ",list1)

List Before pop operation : [1, 2, 3, 4, 5]
List After pop operation : [2, 3, 4, 5]
```

## Difference Between del list1[:] and del list1

list1[:] deletes all elements from the list1 while keeping the reference intact while del list1 deletes the list itself, all elements of list1 along with the reference as demonstrated in the following program:

```
▶ list1=[1,2,3,4,5]
      del list1[:]
      print(list1)

      list2=[1,2,3,4,5]
      del list2
      print(list2)

[]
-----
NameError                                     Traceback (most recent call last)
<ipython-input-20-ead63effa09e> in <module>()
      5 list2=[1,2,3,4,5]
      6 del list2
----> 7 print(list2)

NameError: name 'list2' is not defined

SEARCH STACK OVERFLOW
```

# Conceptualizing Python in Google COLAB

## Deleting an Element of a List Using remove() Method

`remove()` method accepts the name of the element to be deleted. If the element is found, then it is deleted and the remaining elements are re-indexed. If the specified element does not exist in the list, then '**ValueError**' is generated as shown below:

```
▶ list1=[1,2,3,4,5]
  print("List Size Before Deletion :",len(list1))
  print(list1)
  list1.remove(1)
  print(list1[0])
  print("List Size After Deletion  :",len(list1))
  print(list1)

List Size Before Deletion : 5
[1, 2, 3, 4, 5]
2
List Size After Deletion  : 4
[2, 3, 4, 5]
```

```
▶ list1=[1,2,3,4,5]
  print("List Size Before Deletion :",len(list1))
  print(list1)
  list1.remove(10)
  print(list1[0])
  print("List Size After Deletion  :",len(list1))
  print(list1)

List Size Before Deletion : 5
[1, 2, 3, 4, 5]
-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-23-058f0819a63e> in <module>()
      2 print("List Size Before Deletion :",len(list1))
      3 print(list1)
----> 4 list1.remove(10)
      5 print(list1[0])
      6 print("List Size After Deletion  :",len(list1))

ValueError: list.remove(x): x not in list

SEARCH STACK OVERFLOW
```

If the list contains multiple elements with the name passed to `remove()` method, only the first occurrence of the element is deleted. In the following program, the element 1 occurs three times at index positions 0, 2 and 4, respectively. The `remove()` method only deletes the element at index position 0.

# Conceptualizing Python in Google COLAB

```
▶ list1=[1,2,1,4,1]
      print("List Size Before Deletion :",len(list1))
      print(list1)
      list1.remove(1)
      print("List Size After Deletion  :",len(list1))
      print(list1)

↳ List Size Before Deletion : 5
[1, 2, 1, 4, 1]
List Size After Deletion  : 4
[2, 1, 4, 1]
```

For removing all the occurrences of a given element use the following logic: The code iterates through the list and deletes all elements with value 1.

```
▶ list1=[1,2,1,4,1]
      print("List Size Before Deletion :",len(list1))
      print(list1)
      while (True):
          try:
              list1.remove(1)
          except:
              break
      print("List Size After Deletion  :",len(list1))
      print(list1)

      List Size Before Deletion : 5
[1, 2, 1, 4, 1]
List Size After Deletion  : 2
[2, 4]
```

The above program is re-written using lambda function as shown below:

```
▶ list1=[1,2,1,4,1]
      print("List Size Before Deletion :",len(list1))
      print(list1)
      list1=list(filter(lambda x: x != 1, list1))
      print("List Size After Deletion  :",len(list1))
      print(list1)

      List Size Before Deletion : 5
[1, 2, 1, 4, 1]
List Size After Deletion  : 2
[2, 4]
```

# Conceptualizing Python in Google COLAB

## Reversing the List Using reverse() Method

Python supports reverse() method for reversing the elements of a list. The reverse() method does not create a new list but returns the same list with the elements in reverse order.

```
▶ list1=[1,2,3,4,5]
print("List Before reverse operation : ", list1)
list1.reverse()
print("List After reverse operation : ",list1)
```

```
List Before reverse operation :  [1, 2, 3, 4, 5]
List After reverse operation :  [5, 4, 3, 2, 1]
```

As seen by the output generated on execution of the program, the '*id*' of the list before and after reversing is the same. Hence no new list is created.

```
▶ list1=[1,2,3,4,5]
print("id of the list before reversing : ",id(list1))
list2=list1.reverse()
print("id of the list after reversing  : ",id(list1))
print(list2)

id of the list before reversing :  140416040961360
id of the list after reversing  :  140416040961360
None
```

## Finding the index of an Element Using index() Method

Python supports *index()* function which returns the index of the element passed to it. In the following program, the index of the elements ‘PHP’ and ‘Python’ is retrieved using index() method.

```
▶ list1=["C","Python","PHP","Java"]
print("Index of PHP      : ", list1.index("PHP"))
print("Index of Python : ", list1.index("Python"))

Index of PHP      :  2
Index of Python :  1
```

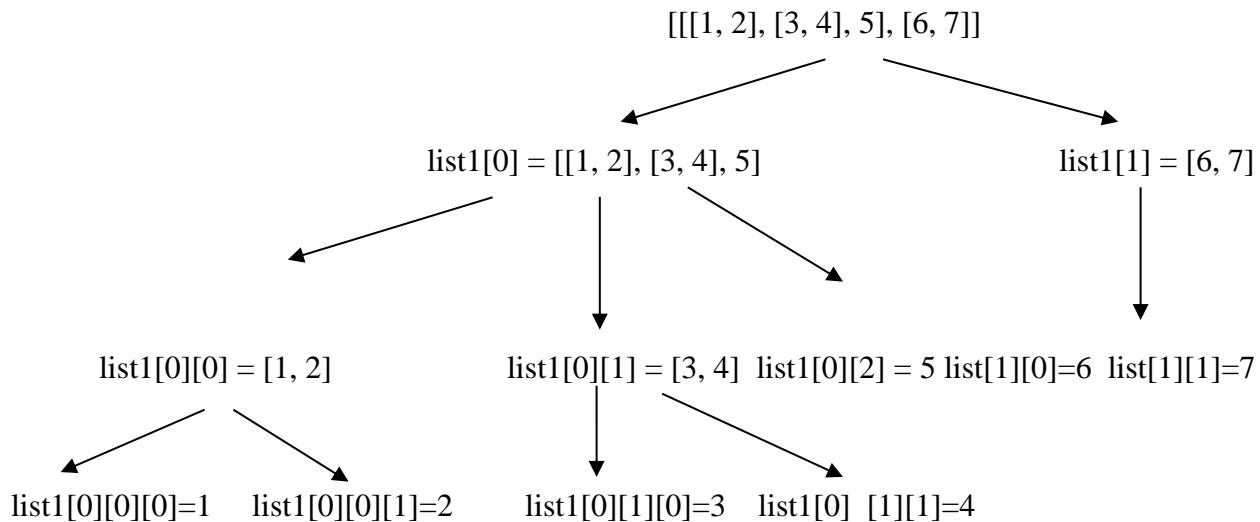
# Conceptualizing Python in Google COLAB

## Nested Lists

A list can be nested inside another list to any level creating a nested list structure. Consider the following list which is nested upto level 3:

```
list1 = [[[1, 2], [3, 4], 5], [6, 7]]
```

The nesting of list1 to different levels is depicted in the following figure:



The following program accesses the different elements of list1 and displays the same:

```
▶ list1 = [[[1, 2], [3, 4], 5], [6, 7]]  
print(list1[0][0][0])  
print(list1[0][0][1])  
print(list1[0][1][0])  
print(list1[0][1][1])  
print(list1[0][2])  
print(list1[1][0])  
print(list1[1][1])  
  
⇨ 1  
2  
3  
4  
5  
6  
7
```

# Conceptualizing Python in Google COLAB

---

## Tuple in Python

A tuple is data structure which contains comma separated heterogeneous elements enclosed within a pair of parentheses () .

### Features of Tuple:

- Tuple is immutable.
- Tuple maintains insertion order.
- Tuple is iterable.
- Tuple is ordered.

### Operations on Tuple

For extracting the elements of a tuple, the slicing operator, positive and negative indexing mechanism discussed above for a list are also applicable to a tuple. The following program employs slicing operator for extracting different elements of a tuple.

```
t=(10, 20.30, "Python", 'c', 23)
t1=('Msc','CSIBER')
print(t[0])
print(t[1:3])
print(t[:2])
print(t[2:])
print(t*2)
print(t+t1)

10
(20.3, 'Python')
(10, 20.3)
('Python', 'c', 23)
(10, 20.3, 'Python', 'c', 23, 10, 20.3, 'Python', 'c', 23)
(10, 20.3, 'Python', 'c', 23, 'Msc', 'CSIBER')
```

# Conceptualizing Python in Google COLAB

---

## Tuple is immutable

Tuple is immutable implies that the elements of a tuple cannot be modified once the tuple is created. An attempt to modify an element of a tuple generates ‘**TypeError**’ as shown in the following figure:

```
▶ t=(10, 20.30, "Python", 'c', 23)
   print(t)
   t[2]="Java"
   print(t)

(10, 20.3, 'Python', 'c', 23)
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-16-3adf4f94e366> in <module>()
      1 t=(10, 20.30, "Python", 'c', 23)
      2 print(t)
----> 3 t[2]="Java"
      4 print(t)

TypeError: 'tuple' object does not support item assignment
```

SEARCH STACK OVERFLOW

However, tuple concatenation is well defined and concatenation of two tuples generates a new tuple as demonstrated in the following program. In the following program, the tuple ‘t’ is concatenated with the tuple (6, 7) which generates a new tuple as confirmed from its ‘id’.

```
▶ t=(1,2,3,4,5)
   print(t)
   print(id(t))
   t=t+(6,7)
   print(id(t))
   print(t)

(1, 2, 3, 4, 5)
140081540476176
140081478906800
(1, 2, 3, 4, 5, 6, 7)
```

# Conceptualizing Python in Google COLAB

---

## Tuple Packing and Unpacking

### Tuple Packing

Tuple packing refers to assigning values to different elements of a tuple in a single statement. The syntax of tuple packing is shown below:

Syntax:

*var = (ele1, ele2, . . . , eleN)*

After the execution of the above statement, the elements of the tuple are assigned to ‘var’ which can be accessed using indices as shown below:

*var[0]=ele1*

*var[1]=ele2*

.

.

*var[n-1]=eleN*

*var1, var2, var3, . . . , varN = value1, value2, value3, . . . , valueN.*

One-to-one correspondence exists between variables and values. In the above syntax, var1, var2, etc. are initialized with value1, value2, ....etc., respectively.

For tuple packing to work properly, the no of values specified on the right side of the expression should exactly be equal to the no. of values on the left hand side of the expression.

```
stud_info=(1,"Maya","A")
print("Roll no  :",stud_info[0])
print("Name    : ",stud_info[1])
print("Division : ",stud_info[2])

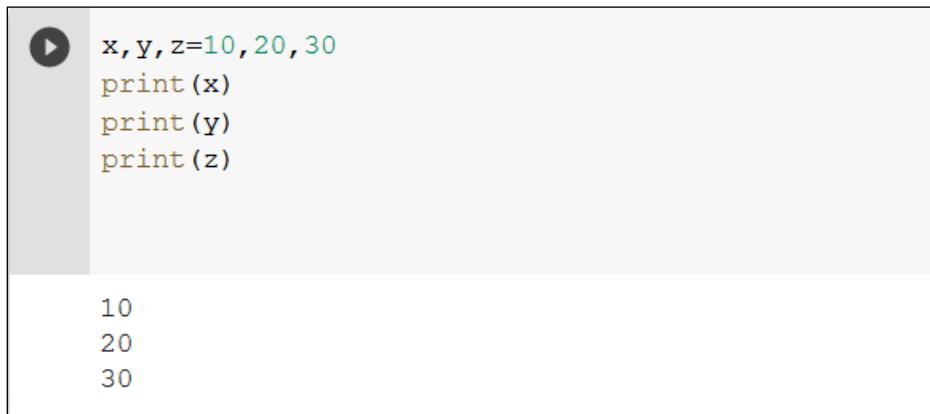
Roll no  : 1
Name    : Maya
Division : A
```

# Conceptualizing Python in Google COLAB

---

## Tuple Unpacking

Tuple unpacking is exact opposite of tuple packing where the list of variables are initialized using the elements of a tuple.



A screenshot of a Google Colab cell. The code cell contains the following Python code:

```
x, y, z=10, 20, 30
print(x)
print(y)
print(z)
```

The output cell shows the results of the print statements:

```
10
20
30
```

If the no. of variables is less than the no. of values, ‘*ValueError*’ is generated with the message ‘*too many values to unpack*’ as shown in the following figure:



A screenshot of a Google Colab cell. The code cell contains the following Python code:

```
x,y,z=10,20,30,40,50
print(x)
print(y)
print(z)
```

The output cell shows the following error message:

```
-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-23-4a8f2a2c4c35> in <module>()
----> 1 x,y,z=10,20,30,40,50
      2     print(x)
      3     print(y)
      4     print(z)
      5

ValueError: too many values to unpack (expected 3)
```

At the bottom of the output cell, there is a button labeled “SEARCH STACK OVERFLOW”.

If the no. of values is less than the no. of variables, ‘*ValueError*’ is generated with the message ‘*not enough values to unpack*’ as shown in the following figure:

# Conceptualizing Python in Google COLAB

```
x,y,z=10,20
print(x)
print(y)
print(z)

-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-24-a5bc6f7212ac> in <module>()
----> 1 x,y,z=10,20
      2 print(x)
      3 print(y)
      4 print(z)
      5

ValueError: not enough values to unpack (expected 3, got 2)

SEARCH STACK OVERFLOW
```

In the following program ‘**stud\_info**’ is a tuple containing student information pertaining to rollno, name and division. The tuple is unpacked for extracting rollno, name and division in three distinct variables.

```
stud_info=(1,"Maya","A")
rollno,name,division=stud_info
print("Roll no : ",rollno)
print("Name : ",name)
print("Division : ",division)

Roll no : 1
Name : Maya
Division : A
```

## Tuple is Iterable

The different elements of the tuple can be traversed using for loop as shown in the following program:

```
t=(1,2,3,4,5)
for ele in t:
    print(ele)

1
2
3
4
5
```

# Conceptualizing Python in Google COLAB

---

## Tuple is Ordered

Two tuples are considered to be equal if and only if they contain same elements also in the same order. In the following program, t1, t2 and t3 contain the same elements. However, t1 and t2 contain the elements in the same order while t1 and t3 contain the same elements in different order. Hence t1 and t2 are considered to be equal while t1 and t3 are not equal as demonstrated in the following example:

```
t1=(1,2,3,4,5)
t2=(1,2,3,4,5)
t3=(5,4,3,2,1)
print(t1==t2)
print(t1==t3)
```

```
True
False
```

```
tuple1=(1,2,3,4,5)
tuple2=(5,4,3,2,1)
tuple3=(1,2,3,4,5)
if (tuple1==tuple2):
    print("tuple1 and tuple2 are equal")
else:
    print("tuple1 and tuple2 are not equal")

if (tuple1==tuple3):
    print("tuple1 and tuple3 are equal")
else:
    print("tuple1 and tuple3 are not equal")
```

```
tuple1 and tuple2 are not equal
tuple1 and tuple3 are equal
```

## Tuple is Hashable

Since the tuple is immutable it can be used as keys of dictionary. Since list is mutable it cannot be used as keys of dictionary. An attempt to use list as a key of a dictionary generates '*TypeError*' as shown in the following figure:

# Conceptualizing Python in Google COLAB



```
1 = [1, 2, 3]
t = (1, 2, 3)
x = {l: 'a list', t: 'a tuple'}
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-1-f1be5784a0c4> in <module>()  
      1 l = [1, 2, 3]  
      2 t = (1, 2, 3)  
----> 3 x = {l: 'a list', t: 'a tuple'}  
  
TypeError: unhashable type: 'list'
```

SEARCH STACK OVERFLOW



```
t1=(1,2,3,4,5)
t2=(1,2,3,4,5)
t3=(5,4,3,2,1)
print(t1.__hash__())
print(t2.__hash__())
print(t3.__hash__())
```

```
8315274433719620810
8315274433719620810
3518382336826571994
```



```
t1=(1,2,3,4,5)
t2=(1,2,3,4,5)
t3=(5,4,3,2,1)
print(id(t1))
print(id(t2))
print(id(t3))
```

```
140081477922448
140081478067408
140081478090096
```

Difference Between id, value and hash

Object id

If the same object is referenced by two distinct references, then their 'id' is same. The 'is' operator compares the 'id' of the two operands. Hence,

# Conceptualizing Python in Google COLAB

---

obj1 is obj2 is equivalent to

`id(obj1) == id(obj2)`.

This is demonstrated in the following program:

```
▶ list1 = [1,2,3]
  list2 = [1,2,3]
  list3 = list1
  print()
  if (id(list1)==id(list2)):
    print("list1 and list2 refer same objects")
  else:
    print("list1 and list2 do not refer same objects")

  if (id(list1)==id(list3)):
    print("list1 and list3 refer same objects")
  else:
    print("list1 and list3 do not refer same objects")
```

⇨ list1 and list2 do not refer same objects  
list1 and list3 refer same objects

The value refers to the object content compared by `==` operator. If the class overrides `__eq__()` method, then `__eq__()` method is invoked when `==` operator is used. If the class does not override `__eq__()` method then the method inherited from `object` class is invoked where the instances will be solely compared using their identities.

```
▶ list1 = [1,2,3]
  list2 = [1,2,3]
  list3 = list1
  print()
  if (list1.__eq__(list2)):
    print("list1 and list2 have same content")
  else:
    print("list1 and list2 do not have same content")

  if (list1.__eq__(list3)):
    print("list1 and list3 have same content")
  else:
    print("list1 and list3 do not have same content")
```

list1 and list2 have same content  
list1 and list3 have same content

# Conceptualizing Python in Google COLAB

---

Some objects are characterized by the hash value which means they can be used as keys in the dictionary. Such objects are characterized by the fact that their value remains same for the life time of the object and as such the objects must be immutable.

**Note:** If you write an `__eq__` method in a custom class, Python will disable this default hash implementation, since your `__eq__` function will define a new meaning of value for its instances. You'll need to write a `__hash__` method as well, if you want your class to still be hashable. If you inherit from a hashable class but don't want to be hashable yourself, you can set `__hash__ = None` in the class body.

`__hash__()` function, `value()` function and equality operator (`==`) return True, if the two objects have the same content, else return False.

On the contrary, `id()` function and ‘is’ operator return True, if the two objects are same, else return False.

## Deleting a Tuple

Tuple does not support deleting a single element. Doing so will generate a ‘**TypeError**’ as shown in the following program.

The screenshot shows a Google Colab cell. The code entered is:

```
t=(1,2,3,4,5)
del t[0]
print(t)
```

The output shows a **TypeError**:

```
TypeError                                 Traceback (most recent call last)
<ipython-input-13-df8988992d1f> in <module>()
      1 t=(1,2,3,4,5)
----> 2 del t[0]
      3 print(t)

TypeError: 'tuple' object doesn't support item deletion
```

Below the output is a button labeled "SEARCH STACK OVERFLOW".

However, ‘**del**’ function can be used for deleting an entire tuple as shown below:

# Conceptualizing Python in Google COLAB

---

```
t=(1,2,3,4,5)
del t
print(t)

-----
NameError Traceback (most recent call last)
<ipython-input-14-d2fd83e9e986> in <module>()
      1 t=(1,2,3,4,5)
      2 del t
----> 3 print(t)

NameError: name 't' is not defined

SEARCH STACK OVERFLOW
```

## Operation Not Applicable to Tuple

Since tuple is immutable, the following operations cannot be performed on a tuple:

- append()
- insert()
- clear()
- pop()
- remove()
- copy()
- sort()
- reverse()

## Tuple Nested Inside List

A tuple can be nested inside list. Tuple being immutable cannot be changed, however, the list can be manipulated. An attempt to change the tuple element of a list element generates '**TypeError**' as shown in the following figure:

# Conceptualizing Python in Google COLAB

```
▶ students=[(1,"Maya","A"),(2,"Milan","B"),(3,"Asha","C")]
  students[0][1]="Sachin"
  print(students[0][1])

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-17-46e1b841162d> in <module>()
      1 students=[(1,"Maya","A"),(2,"Milan","B"),(3,"Asha","C")]
----> 2 students[0][1]="Sachin"
      3 print(students[0][1])

TypeError: 'tuple' object does not support item assignment
```

[SEARCH STACK OVERFLOW](#)

However, the list element can be directly manipulated as shown below:

```
▶ students=[(1,"Maya","A"),(2,"Milan","B"),(3,"Asha","C")]
  students[0]=(1,"Sachin","A")
  print(students)

[(1, 'Sachin', 'A'), (2, 'Milan', 'B'), (3, 'Asha', 'C')]
```

## List Nested Inside Tuple

A list can be nested inside a tuple. Tuple being immutable cannot be changed, however, the list can be manipulated. An attempt to change the tuple element generates '**TypeError**' as shown in the following figure:

```
▶ students=([(1,"Maya","A"),[2,"Milan","B"],[3,"Asha","C"]])
  students[0]=[1,"Sachin","A"]
  print(students)

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-20-ba4fe87aeff1> in <module>()
      1 students=([(1,"Maya","A"),[2,"Milan","B"],[3,"Asha","C"]])
----> 2 students[0]=[1,"Sachin","A"]
      3 print(students)

TypeError: 'tuple' object does not support item assignment
```

[SEARCH STACK OVERFLOW](#)

However, the list element can be directly manipulated as shown below:

# Conceptualizing Python in Google COLAB

---

```
▶ students=[[1,"Maya","A"],[2,"Milan","B"],[3,"Asha","C"])
students[0][1]="Sachin"
print(students)
```

```
([1, 'Sachin', 'A'], [2, 'Milan', 'B'], [3, 'Asha', 'C'])
```

## Dictionary in Python

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). They are similar to the hash tables containing key/value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

## Operations on Dictionary

### Accessing Keys of a Dictionary

'dict' class supports the methods keys() and values() which return a tuple containing list of keys and a tuple containing list of values in a dictionary as shown in the following program:

```
▶ dict={}
dict["course"]="M.Sc"
dict[2]="Two"
print(dict)
print(dict["course"])
print(dict[2])
print(dict.keys())
print(dict.values())
```

```
{'course': 'M.Sc', 2: 'Two'}
M.Sc
Two
dict_keys(['course', 2])
dict_values(['M.Sc', 'Two'])
```

# Conceptualizing Python in Google COLAB

---

## fromkeys() Method of dict class

*fromkeys()* method of ‘*dict*’ class can be used for creating the elements of a dictionary as demonstrated in the following program. The *fromkeys()* method accepts two parameters, the first parameter is the list containing keys and the second parameter is a value.

```
▶ students={}
   print(students)

{'rollno': 1}
```

## Iterating Over the Elements of Dictionary

There are three approaches for iterating over the elements of a dictionary.

### Method 1:

Using the *keys()* method of a ‘*dict*’ class.

```
▶ student={"rollno":1,"name":"Maya","division":"A"}
   for k in student.keys():
       print(student[k])

1
Maya
A
```

### Method 2:

Using the *values()* method of a ‘*dict*’ class.

# Conceptualizing Python in Google COLAB

---

```
student={"rollno":1,"name":"Maya","division":"A"}  
for v in student.values():  
    print(v) |
```

```
1  
Maya  
A
```

## Method 3:

Using the items() method of a '*dict*' class. The items() method of 'dict' class returns key/value pair of each element in the dictionary.

```
student={"rollno":1,"name":"Maya","division":"A"}  
for (k,v) in student.items():  
    print(k,v)
```

```
rollno 1  
name Maya  
division A
```

## Dictionary is Mutable

In a dictionary the key is unique. The values of the keys can be altered after the dictionary is created.

```
student={"rollno":1,"name":"Maya","division":"A"}  
print(student)  
student["rollno"]=2  
student["name"]="Milan"  
print(student)
```

```
{'rollno': 1, 'name': 'Maya', 'division': 'A'}  
{'rollno': 2, 'name': 'Milan', 'division': 'A'}
```

# Conceptualizing Python in Google COLAB

## Concatenation of Two Dictionaries

The concatenation operation on two dictionaries is not defined. An attempt to add two dictionaries generates '*TypeError*' as shown in the following program:

```
▶ students={"rollno":1,"name":"Maya","division":"A"}  
student2={"rollno":2,"name":"Milan","division":"B"}  
students=students+student2  
  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-35-55fb4c94887d> in <module>()  
  1 students={"rollno":1,"name":"Maya","division":"A"}  
  2 student2={"rollno":2,"name":"Milan","division":"B"}  
----> 3 students=students+student2  
  
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

[SEARCH STACK OVERFLOW](#)

## update() Method

The *update()* method can be used for updating a dictionary based on the contents of another dictionary passed to it as an argument.

```
▶ students={"rollno":1,"name":"Maya","division":"A"}  
student2={"rollno":2,"name":"Milan","division":"B"}  
students.update(student2)  
print(students)  
  
{'rollno': 2, 'name': 'Milan', 'division': 'B'}
```

If the key is already present in the dictionary, the corresponding value is updated, otherwise a new key/value pair is added to the dictionary as shown below:

```
▶ students={"rollno":1,"name":"Maya"}  
students1={"rollno":2,"division":"A"}  
print("Before Updation : ",students)  
students.update(students1)  
print("After Updation : ",students)  
  
Before Updation : {'rollno': 1, 'name': 'Maya'}  
After Updation : {'rollno': 2, 'name': 'Maya', 'division': 'A'}
```

# Conceptualizing Python in Google COLAB

---

## Membership Testing in Dictionary

The '**in**' operator is used for testing the existence of a key n a dictionary as demonstrated in the following program:

```
student={"rollno":1,"name":"Maya","division":"A"}  
if ("division" in student):  
    print(student["division"])  
else:  
    print("Key does not exist")
```

A

## Deleting Elements of Dictionary

### Method 1: Using pop() method

**pop()** method of a dict class accepts the key of the dictionary element to be deleted as demonstrated in the following program:

```
student={"rollno":1,"name":"Maya","division":"A"}  
print(student)  
if ("division" in student):  
    student.pop("division")  
else:  
    print("Key does not exist")  
print(student)
```

{'rollno': 1, 'name': 'Maya', 'division': 'A'}  
{'rollno': 1, 'name': 'Maya'}

### Method 2: Using del Method

del method can also be used for deleting the element of a dictionary as shown below:

# Conceptualizing Python in Google COLAB

```
student={"rollno":1,"name":"Maya","division":"A"}  
print(student)  
if ("division" in student):  
    del student["division"]  
else:  
    print("Key does not exist")  
print(student)  
  
{'rollno': 1, 'name': 'Maya', 'division': 'A'}  
{'rollno': 1, 'name': 'Maya'}
```

## Set in Python

Similar to dictionary, {} characters are used for enclosing the elements of a set. A set can contain only immutable elements. Adding mutable elements to a set generates '**TypeError**' as demonstrated in the following programs:

```
s=[[1,2],3,4,5]  
  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-1-c5647b504393> in <module>()  
----> 1 s=[[1,2],3,4,5]  
  
TypeError: unhashable type: 'list'
```

SEARCH STACK OVERFLOW

```
s={{"one":1},3,4,5}  
  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-3-27abaeaf8462> in <module>()  
----> 1 s={{"one":1},3,4,5}
```

TypeError: unhashable type: 'dict'

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

---

```
 s={(1,2),3,4,5}  
print(s)  
  
{(1, 2), 3, 4, 5}
```

## Features of Set:

- Set is immutable.
- Set elements are unique
- Set elements are not indexed.
- Set does not maintain insertion order.
- Set is iterable.
- Set is not ordered.

## Set is Immutable

Set is immutable implies that the elements of a set cannot be modified once the set is created. An attempt to modify an element of a tuple generates '**TypeError**' as shown in the following figure:

```
 s={(1,2),3,4,5}  
s[0]=1  
  
-----  
TypeError: 'set' object does not support item assignment                                         Traceback (most recent call last)  
<ipython-input-6-d3d9429d76ac> in <module>()  
      1 s={(1,2),3,4,5}  
----> 2 s[0]=1  
  
TypeError: 'set' object does not support item assignment  
  
SEARCH STACK OVERFLOW
```

# Conceptualizing Python in Google COLAB

---

## Set Elements are Unique

Addition of any duplicate element to the set is discarded by the set as shown in the following example:

```
▶ s={1,2,3,1,2,3}
      print(s)

{1, 2, 3}
```

## Set Elements are not Indexed – Set Does not Maintain Insertion Order

Displaying the unsorted set multiple times displays the elements in a different order each time. If the set is sorted the order is maintained as shown below:

```
▶ s={1,2,3,4,5,6,7,8}
      print(s)
      print(s)

{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 7, 8}
```

The order of elements in a set is unpredictable.

```
▶ s={11,2,34,5,12,67,55}
      print(s)

{2, 34, 67, 5, 11, 12, 55}
```

Set elements are not indexed. An attempt to access the elements of a set using indexing generates '**TypeError**' as shown in the following program:

# Conceptualizing Python in Google COLAB

```
s1={1,2,3}
print(s1[0])
```

-----

```
TypeError                                     Traceback (most recent call last)
<ipython-input-17-a5a8309882c8> in <module>()
      1 s1={1,2,3}
----> 2 print(s1[0])

TypeError: 'set' object is not subscriptable
```

[SEARCH STACK OVERFLOW](#)

## Set is Iterable

The different elements of the set can be traversed using for loop as shown in the following program:

```
s1={1,2,3}
for ele in s1:
    print(ele)
```

1  
2  
3

## Concatenating Two Sets

The concatenation operation on two sets is not defined. An attempt to add two sets generates '**TypeError**' as shown in the following program:

Since

```
s1={1,2,3}
s2={4,5,6}
s3=s1+s2
print(s3)
```

-----

```
TypeError                                     Traceback (most recent call last)
<ipython-input-14-50a40a8a7a55> in <module>()
      1 s1={1,2,3}
      2 s2={4,5,6}
----> 3 s3=s1+s2
      4 print(s3)

TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

[SEARCH STACK OVERFLOW](#)

# Conceptualizing Python in Google COLAB

---

Since the concatenation operation is defined on two lists, for concatenating two sets performs the following operations:

- Convert the two sets into lists using `list()` method
- Concatenate the two lists
- Convert the list back into a set

as demonstrated in the following program:

```
s1={1,2,3}  
print("Set 1",s1)  
s2={4,5,6}  
print("Set 2",s2)  
s3=set(list(s1)+list(s2))  
print("Set 3",s3)
```

```
Set 1 {1, 2, 3}  
Set 2 {4, 5, 6}  
Set 3 {1, 2, 3, 4, 5, 6}
```

## Sorting a Set

Since the order of the elements in a set is insignificant, sorting and reversing the elements of a set do not create new sets. However, `sorted()` methods is supported while `reversed()` method is not supported on a set, `sorted()` method returns list as demonstrated in the following program:

```
s=[1,24,3,44,33]  
print("Unsorted Set : ",s)  
print("Sorted Set : ",sorted(s))  
print(type(sorted(s)))
```

```
Unsorted Set :  [1, 33, 3, 44, 24]  
Sorted Set :  [1, 3, 24, 33, 44]  
<class 'list'>
```

## Set is Not Reversible

Since the set is immutable, the elements of a set cannot be reversed. Doing so generates '`TypeError`' as demonstrated in the following program:

# Conceptualizing Python in Google COLAB

```
s={1,24,3,44,33}
print("Original Set : ",s)
print("Reversed Set : ",reversed(s))
print(type(revesed(s)))

Original Set : {1, 33, 3, 44, 24}
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-2-ddd6abe7549b> in <module>()
      1 s={1,24,3,44,33}
      2 print("Original Set : ",s)
----> 3 print("Reversed Set : ",reversed(s))
      4 print(type(revesed(s)))

TypeError: 'set' object is not reversible
```

[SEARCH STACK OVERFLOW](#)

## Checking Set Equality Using == Operator

The equality operator == can be used for comparing the content of two sets. == returns '**True**' if the two sets have same content, otherwise '**False**'.

```
s1={1,2, 3, 4, 5}
s2={1,2, 3, 4, 5}
s3=s1
if (s1==s2):
    print("s1 is equal to s2")
else:
    print("s1 is not equal to s2")
if (s1==s3):
    print("s1 is equal to s3")
else:
    print("s1 is not equal to s3")
```

↳ s1 is equal to s2  
s1 is equal to s3

## Checking Set Equivalence Using 'is' Operator

'is' operator returns '**True**' if the two references reference the same set object, otherwise returns '**False**'.

# Conceptualizing Python in Google COLAB

```
s1={1,2, 3, 4, 5}
s2={1,2, 3, 4, 5}
s3=s1
if (s1 is s2):
    print("s1 is s2")
else:
    print("s1 is not s2")
if (s1 is s3):
    print("s1 is s3")
else:
    print("s1 is not s3")

s1 is not s2
s1 is s3
```

## Inserting a Single Element in a List – add() Method

If the list is sorted, then the add() method inserts the element at the appropriate position as demonstrated in the following figures:

```
s1={1,2, 3, 4, 5,6, 7, 8, 9, 10}
print("Before Insertion", s1)
s1.add(100)
print("After Insertion", s1)

Before Insertion {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
After Insertion {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100}
```

```
s1={1,2, 3, 4, 5,6, 7, 8, 9, 50}
print("Before Insertion", s1)
s1.add(40)
print("After Insertion", s1)

Before Insertion {1, 2, 3, 4, 5, 6, 7, 8, 9, 50}
After Insertion {1, 2, 3, 4, 5, 6, 7, 8, 9, 40, 50}
```

If the set is unsorted, then add() method inserts the element at random position as shown below:

# Conceptualizing Python in Google COLAB



```
s1={1,20, 3, 45, 15,6, 57, 33, 11}
print("Before Insertion", s1)
s1.add(100)
print("After Insertion", s1)

Before Insertion {1, 33, 3, 6, 11, 45, 15, 20, 57}
After Insertion {1, 33, 3, 100, 6, 11, 45, 15, 20, 57}
```

## Inserting a Multiple Elements in a List – update() Method

For inserting multiple elements in the set *update()* method can be employed which accepts an iterable (list, tuple or set) as its single argument as demonstrated in the following programs:

### Using list to insert elements in a set

The update() method accepts an iterable, iterates through all the elements and adds each element to the set. In the following example, the list containing five elements is passed to update() element which inserts all the elements of a list to a set.



```
s1={1, 2, 3, 4, 5}
print("Before Insertion", s1)
s1.update([6, 7, 8, 9, 10])
print("After Insertion", s1)

Before Insertion {1, 2, 3, 4, 5}
After Insertion {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

### Using tuple to insert elements in a set

In the following example, the tuple containing five elements is passed to update() element which inserts all the elements of a tuple to a set.



```
s1={1, 2, 3, 4, 5}
print("Before Insertion", s1)
s1.update((6, 7, 8, 9, 10))
print("After Insertion", s1)

Before Insertion {1, 2, 3, 4, 5}
After Insertion {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

# Conceptualizing Python in Google COLAB

## Using set to insert elements in a set

In the following example, the set containing five elements is passed to update() element which inserts all the elements of a target set to a source set.

```
▶ s1={1, 2, 3, 4, 5}
    print("Before Insertion", s1)
    s1.update({6, 7, 8, 9, 10})
    print("After Insertion", s1)

Before Insertion {1, 2, 3, 4, 5}
After Insertion {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

## Deleting an Element from a Set – remove() Method

```
▶ s1={1, 2, 3, 4, 5}
    print("Before Deletion", s1)
    s1.remove(3)
    print("After Deletion", s1)

⇒ Before Deletion {1, 2, 3, 4, 5}
    After Deletion {1, 2, 4, 5}
```

If the element does not exist in a set, then remove() method generates ‘KeyError’ as demonstrated in the following program:

```
▶ s1={1, 2, 3, 4, 5}
    print("Before Deletion", s1)
    s1.remove(6)
    print("After Deletion", s1)

Before Deletion {1, 2, 3, 4, 5}
-----
KeyError                                                 Traceback (most recent call last)
<ipython-input-15-38c9f99a673c> in <module>()
      1 s1={1, 2, 3, 4, 5}
      2 print("Before Deletion", s1)
----> 3 s1.remove(6)
      4 print("After Deletion", s1)

KeyError: 6
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

---

## Deleting an Element from a Set – discard() Method

```
s1={1, 2, 3, 4, 5}
print("Before Deletion", s1)
s1.discard(3)
print("After Deletion", s1)
```

```
Before Deletion {1, 2, 3, 4, 5}
After Deletion {1, 2, 4, 5}
```

If the element does not exist in a set, then unlike remove() method discard() method does not generate any error as demonstrated in the following program:

```
s1={1, 2, 3, 4, 5}
print("Before Deletion", s1)
s1.discard(6)
print("After Deletion", s1)
```

```
Before Deletion {1, 2, 3, 4, 5}
After Deletion {1, 2, 3, 4, 5}
```

## Deleting an Element from a Set – pop() Method

pop() method does not accept any arguments and deletes the first element from the list and returns the deleted element as shown in the following program:

```
s1={1, 2, 3, 4, 5}
print("Before Deletion", s1)
ele=s1.pop()
print("After Deletion of element ", ele, " set is ", s1)
```

```
Before Deletion {1, 2, 3, 4, 5}
After Deletion of element 1 set is {2, 3, 4, 5}
```

remove() and discard() methods do not return any value.

# Conceptualizing Python in Google COLAB

---

```
s1={1, 2, 3, 4, 5}
print("Before Deletion ", s1)
print(s1.discard(3))
print("After Deletion ", s1)
```

```
Before Deletion {1, 2, 3, 4, 5}
None
After Deletion {1, 2, 4, 5}
```

```
s1={1, 2, 3, 4, 5}
print("Before Deletion ", s1)
print(s1.remove(3))
print("After Deletion ", s1)
```

```
Before Deletion {1, 2, 3, 4, 5}
None
After Deletion {1, 2, 4, 5}
```

## Deleting all the Elements from a Set – clear() Method

```
s1={1, 2, 3, 4, 5}
print("Before Deletion ", s1)
print(s1.clear())
print("After Deletion ", s1)
```

```
Before Deletion {1, 2, 3, 4, 5}
None
After Deletion set()
```

clear() method does not return any value.

# Conceptualizing Python in Google COLAB

## Deleting a set Using del Method

Since set is not indexed, del method cannot be used for deleting the elements of a set. Doing so generates '*TypeError*' as shown in the following program:

```
s1={1, 2, 3, 4, 5}
print("Before Deletion ", s1)
del s1[:]
print("After Deletion ", s1)

Before Deletion {1, 2, 3, 4, 5}
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-27-9b480505f625> in <module>()
      1 s1={1, 2, 3, 4, 5}
      2 print("Before Deletion ", s1)
----> 3 del s1[:]
      4 print("After Deletion ", s1)

TypeError: 'set' object does not support item deletion
```

SEARCH STACK OVERFLOW

'*del*' method can however be used for deleting the entire set as shown in the following program:

```
s1={1, 2, 3, 4, 5}
print("Before Deletion ", s1)
del s1
print("After Deletion ", s1)

Before Deletion {1, 2, 3, 4, 5}
-----
NameError                                     Traceback (most recent call last)
<ipython-input-28-5e2b5beea38d> in <module>()
      2 print("Before Deletion ", s1)
      3 del s1
----> 4 print("After Deletion ", s1)

NameError: name 's1' is not defined
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

---

## Cloning Set

```
▶ s1={(1, 2, 3), (4, 5, 6)}
      s2=s1.copy()
      print("s1 : ", s1)
      print("s2 : ", s2)
      s1.update((7, 8, 9))
      print("s1 : ", s1)
      print("s2 : ", s2)

▶ s1 : {(4, 5, 6), (1, 2, 3)}
      s2 : {(4, 5, 6), (1, 2, 3)}
      s1 : {7, 8, 9, (1, 2, 3), (4, 5, 6)}
      s2 : {(4, 5, 6), (1, 2, 3)}
```

## Set Aggregate Functions –max(), min(), sum()

The aggregate functions max(), min(), sum() can be used for finding the maximum, minimum and sum of all elements of a set as demonstrated in the following program:

```
▶ s1={1,2,3,4,5}
      print("Maximum : ",max(s1))
      print("Minimum : ",min(s1))
      print("Sum      : ",sum(s1))

      Maximum : 5
      Minimum : 1
      Sum      : 15
```

```
▶ s1={(10, 2, 2), (4, 5, 6)}
      print(max(s1))

      (10, 2, 2)
```

# Conceptualizing Python in Google COLAB

---

## Set Operations

Different set operations such as union, intersection, set difference, set symmetric difference, etc. are defined on two sets.

### Union of Two Sets

Union of two sets is a set containing elements from both the sets eliminating duplicates, if any as shown in the following program. Set supports union() method for the purpose. The same result can be achieved using ‘|’ operator as well.

```
▶ s1={1,2,3,4,5}
  s2={3,4,5,6,7}
  print("s1 : ",s1)
  print("s2 : ",s2)
  s3=s1.union(s2)
  s4=s1 | s2
  print("s1 U s2 : ",s3)
  print("s1 U s2 : ",s4)

  s1 : {1, 2, 3, 4, 5}
  s2 : {3, 4, 5, 6, 7}
  s1 U s2 : {1, 2, 3, 4, 5, 6, 7}
  s1 U s2 : {1, 2, 3, 4, 5, 6, 7}
```

### Intersection of Two Sets

Intersection of two sets is a set containing elements common to both the sets as shown in the following program. Set supports intersection() method for the purpose. The same result can be achieved using ‘&’ operator as well.

```
▶ s1={1,2,3,4,5}
  s2={3,4,5,6,7}
  print("s1 : ",s1)
  print("s2 : ",s2)
  s3=s1.intersection(s2)
  s4=s1 & s2
  print("s1 & s2 : ",s3)
  print("s1 & s2 : ",s4)

  s1 : {1, 2, 3, 4, 5}
  s2 : {3, 4, 5, 6, 7}
  s1 & s2 : {3, 4, 5}
  s1 & s2 : {3, 4, 5}
```

# Conceptualizing Python in Google COLAB

---

## Set Difference

Difference of two sets is a set containing elements from the first set which are not present in second set as shown in the following program. Set supports difference() method for the purpose. The same result can be achieved using ‘-’ operator as well.

```
s1={1,2,3,4,5}
s2={3,4,5,6,7}
print("s1 : ",s1)
print("s2 : ",s2)
s3=s1.difference(s2)
s4=s2.difference(s1)
print("s1 - s2 : ",s3)
print("s2 - s1 : ",s4)

s1 : {1, 2, 3, 4, 5}
s2 : {3, 4, 5, 6, 7}
s1 - s2 : {1, 2}
s2 - s1 : {6, 7}
```

## Set Symmetric Difference

Symmetric difference of two sets is a set containing elements from both the sets obtained by computing their respective differences as shown in the following program. Set supports symmetric\_difference() method for the purpose. The same result can be achieved using the following operation:

$$s1 \sim s2 = s1 - s2 \cup (s2 - s1)$$

```
s1={1,2,3,4,5}
s2={3,4,5,6,7}
print("s1 : ",s1)
print("s2 : ",s2)
s3=s1.symmetric_difference(s2)
s4=s2.symmetric_difference(s1)
print("(s1 - s2) U (s2 - s1) : ",s3)
print("(s2 - s1) U (s1 - s2) : ",s4)

s1 : {1, 2, 3, 4, 5}
s2 : {3, 4, 5, 6, 7}
(s1 - s2) U (s2 - s1) : {1, 2, 6, 7}
(s2 - s1) U (s1 - s2) : {1, 2, 6, 7}
```

# Conceptualizing Python in Google COLAB

---

## isdisjoint() Method

The two sets are considered to be disjoint, if they have no elements in common. set class supports a boolean method isdisjoint() to check whether the two sets are disjoint or not.

```
s1={1, 2, 3}
s2={3, 4, 5}
s3={4, 5, 6}
print("s1 and s2 are disjoint : ",s1.isdisjoint(s2))
print("s1 and s3 are disjoint : ",s1.isdisjoint(s3))

s1 and s2 are disjoint :  False
s1 and s3 are disjoint :  True
```

## Checking Set Relationship – issubset(), issuperset()

Set A is considered to be a subset of set B, if it at least contains all elements of set B. B is considered to be a superset of A. set class supports two boolean methods issubset() and issuperset() to check whether the invoking set is a subset/superset of a set passed as an argument.

```
s1={1, 2, 3}
s2={1, 2}
print("s2 is subset of s1    : ",s2.issubset(s1))
print("s1 is superset of s2 : ",s1.issuperset(s2))

s2 is subset of s1    :  True
s1 is superset of s2 :  True
```

```
s1={1, 2, 3}
s2={1, 2, 5}
print("s2 is subset of s1    : ",s2.issubset(s1))
print("s1 is superset of s2 : ",s1.issuperset(s2))

s2 is subset of s1    :  False
s1 is superset of s2 :  False
```

## Set Difference Revisited- difference() Method

Set difference contains all the elements from first set which are not present in second set, but it does not update the first set as shown below:

# Conceptualizing Python in Google COLAB

---

```
s1={1,2,3}
s2={1,2,5}
s3=s1.difference(s2)
print("s1 : ",s1)
print("s2 : ",s2)
print("s3 : ",s3)

s1 : {1, 2, 3}
s2 : {1, 2, 5}
s3 : {3}
```

## ***difference\_update()* Method**

*difference\_update()* method on the contrary computes the set difference and updates the invoking set.

```
s1={1,2,3}
s2={1,2,5}
s3=s1.difference_update(s2)
print("s1 : ",s1)
print("s2 : ",s2)
print("s3 : ",s3)

s1 : {3}
s2 : {1, 2, 5}
s3 : None
```

## ***intersection\_update()* Method**

*intersection\_update()* method computes the set intersection and updates the invoking set with the result of set intersection.

```
s1={1,2,3}
s2={1,2,5}
s3=s1.intersection_update(s2)
print("s1 : ",s1)
print("s2 : ",s2)
print("s3 : ",s3)

s1 : {1, 2}
s2 : {1, 2, 5}
s3 : None
```

# Conceptualizing Python in Google COLAB

---

## Symmetric\_difference\_update() Method

*Symmetric\_difference\_update()* method computes the symmetric difference between two sets and updates the invoking set with the result of set difference.

```
▶ s1={1,2,3}
  s2={1,2,5}
  s3=s1.symmetric_difference_update(s2)
  print("s1 : ",s1)
  print("s2 : ",s2)
  print("s3 : ",s3)

  s1 : {3, 5}
  s2 : {1, 2, 5}
  s3 : None
```

## Frozen Set

Set is immutable. Frozen set is a mutable version of a set. Since the set is mutable, it is not hashable and hence cannot be used as a key of a dictionary whereas a frozen set can be.

## Converting a Set into Frozen Set

A set can be converted into a frozen set using *frozenset()* method as demonstrated in the following program. The program confirms that set is not hashable while frozen set is. After converting a set into a frozen set, the methods which modify the elements of a set such as *add()*, *clear()*, *update()* etc. case to be applicable to a frozen set.

```
▶ set1 = {1,2,3}
try:
    print(hash(set1))
except TypeError:
    print("Set is not Hashable")

fset1=frozenset(set1)
try:
    print("Hash Value of Frozen Set : ",hash(fset1))
except TypeError:
    print("Object is not Hashable")

Set is not Hashable
Hash Value of Frozen Set : -272375401224217160
```

# Conceptualizing Python in Google COLAB

## Application of Set

To find difference between two lists.

The List class does not support ‘-‘ operator for finding the difference between two lists, however Set class does.

```
▶ list1=[1,2,3,4,5]
  list2=[1,3,5]
  list3=list1-list2
  print(list1)
  print(list2)
  print(list3)

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-14-12565556f06f> in <module>()
      1 list1=[1,2,3,4,5]
      2 list2=[1,3,5]
----> 3 list3=list1-list2
      4 print(list1)
      5 print(list2)

TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

[SEARCH STACK OVERFLOW](#)

Hence using Set class the difference between the elements of ‘list1’ and ‘list2’ can be computed as demonstrated in the following program:

```
▶ list1=[1,2,3,4,5]
  list2=[1,3,5]
  list3=list(set(list1)-set(list2))
  print(list1)
  print(list2)
  print(list3)

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

# Conceptualizing Python in Google COLAB

---

## Chapter 3

### Lab Assignment on Python Operators and Control Statements

#### Level – Basic

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

### Python Arithmetic Operators

The different arithmetic operators supported by Python language are depicted in the following table:

Assume variable a holds 10 and variable b holds 20, then –

# Conceptualizing Python in Google COLAB

Operator	Description	Example
+	Addition Adds values on either side of the operator.	$a + b = 30$
-	Subtraction Subtracts right hand operand from left hand operand.	$a - b = -10$
*	Multiplication Multiplies values on either side of the operator	$a * b = 200$
/	Division Divides left hand operand by right hand operand	$b / a = 2$
%	Modulus Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
**	Exponent Performs exponential (power) calculation on operators	$a^{**}b = 10 \text{ to the power } 20$
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	$9//2 = 4 \text{ and } 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0$

## Floor Division Operator

Python supports floor division operator which performs an integer division as shown in the following program:

```
▶ a=30
   b=20
   print(a/b)
   print(a//b)

1.5
1
```

# Conceptualizing Python in Google COLAB

---

## Python Assignment Operators

The different assignment operators supported by Python language are depicted in the following table:

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c // a is equivalent to c = c // a

## Python Comparison Operators

The comparison or relational operators compare the values of two operands. The different relational operators supported by Python language are depicted in the following table:

Assume variable a holds 10 and variable b holds 20, then –

# Conceptualizing Python in Google COLAB

---

Operator	Description	Example
<code>==</code>	If the values of two operands are equal, then the condition becomes true.	<code>(a == b)</code> is not true.
<code>!=</code>	If values of two operands are not equal, then condition becomes true.	<code>(a != b)</code> is true.
<code>&lt;&gt;</code>	If values of two operands are not equal, then condition becomes true.	<code>(a &lt;&gt; b)</code> is true. This is similar to <code>!=</code> operator.
<code>&gt;</code>	If the value of left operand is greater than the value of right operand, then condition becomes true.	<code>(a &gt; b)</code> is not true.
<code>&lt;</code>	If the value of left operand is less than the value of right operand, then condition becomes true.	<code>(a &lt; b)</code> is true.
<code>&gt;=</code>	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	<code>(a &gt;= b)</code> is not true.
<code>&lt;=</code>	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	<code>(a &lt;= b)</code> is true.

## Python Bitwise Operators

The different bitwise operators supported by Python language are depicted in the following table:

Operator	Description	Example
<code>&amp; Binary AND</code>	Operator copies a bit to the result if it exists in both operands	<code>(a &amp; b)</code> (means 0000 1100)
<code>  Binary OR</code>	It copies a bit if it exists in either operand.	<code>(a   b) = 61</code> (means 0011 1101)
<code>^ Binary XOR</code>	It copies the bit if it is set in one operand but not both.	<code>(a ^ b) = 49</code> (means 0011 0001)
<code>~ Binary Ones Complement</code>	It is unary and has the effect of 'flipping' bits.	<code>(~a) = -61</code> (means 1100 0011 in 2's complement form due to a signed binary number.)
<code>&lt;&lt; Binary Left Shift</code>	The left operand's value is moved left by the number of bits specified by the right operand.	<code>a &lt;&lt; 2 = 240</code> (means 1111 0000)
<code>&gt;&gt; Binary Right Shift</code>	The left operand's value is moved right by the number of bits specified by the right operand.	<code>a &gt;&gt; 2 = 15</code> (means 0000 1111)

# Conceptualizing Python in Google COLAB

---

Bitwise operator works on bits and performs bit by bit operation. Assume if  $a = 60$ ; and  $b = 13$ ; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands –

	128	64	32	16	8	4	2	1
60	0	0	1	1	1	1	0	0
13	0	0	0	0	1	1	0	1
60 & 13 =12	0	0	0	0	1	1	0	0
60   13 =61	0	0	1	1	1	1	0	1
60 ^ 13 =49	0	0	1	1	0	0	0	1

$$a = 0011\ 1100$$

$$b = 0000\ 1101$$

$$a \& b = 0000\ 1100$$

$$a|b = 0011\ 1101$$

$$a^b = 0011\ 0001$$

$$\sim a = 1100\ 0011$$

The different bitwise operations on operands 60 and 13 are demonstrated in the following program:

# Conceptualizing Python in Google COLAB

```
a = 60  
b = 13  
print(a & b)  
print(a | b)  
print(a ^ b)  
print(~a)
```

```
12  
61  
49  
-61
```

## Left Shift and Right Shift Operators

```
a = 8  
print(a >> 1)  
print(a << 1)
```

```
4  
16
```

## Python Logical Operators

The different logical operators supported by Python language are depicted in the following table:

Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

# Conceptualizing Python in Google COLAB

---

## Short-Circuit Evaluation

### In Evaluation of ‘and’ Condition

The following program demonstrates short-circuit evaluation of ‘**and**’ condition in Python. In the following program, in the ‘**if**’ condition, the first condition evaluates to true, hence the second condition is evaluated which invokes *display()* method which prints ‘**Inside display**’ as shown in the following program:

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

x=20
if (x>10 and display()):
    print("TRUE")
```

Inside display

When the value of ‘x’ is changed to 2, the first condition in the ‘**if**’ statement evaluates to false and hence the second condition is not evaluated owing to short-circuit evaluation adopted by Python.

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

x=2
if (x>10 and display()):
    print("TRUE")
```

### In Evaluation of ‘or’ Condition

The following program demonstrates short-circuit evaluation of ‘**or**’ condition in Python. In the following program, in the ‘**if**’ condition, the first condition evaluates to false, hence the second condition is evaluated which invokes *display()* method which prints ‘**Inside display**’ as shown in the following program:

# Conceptualizing Python in Google COLAB

---

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

x=2
if (x>10 or display()):
    print("TRUE")
```

Inside display

When the value of ‘`x`’ is changed to 20, the first condition in the ‘`if`’ statement evaluates to true and hence the second condition is not evaluated owing to short-circuit evaluation adopted by Python.

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

x=20
if (x>10 or display()):
    print("TRUE")
```

TRUE

```
def display():
    print("Inside display")
    return True

x=2
if (x>10 or display()):
    print("TRUE")
```

Inside display

TRUE

# Conceptualizing Python in Google COLAB

```
def display():
    print("Inside display")
    return False

x=2
if (x>10 or display()):
    print("TRUE")
```

Inside display

## Python Membership Operators

Python's membership operators are employed to test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below –

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not find a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

The following programs demonstrate the use of membership operators in Python:

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

True

```
x = ["apple", "banana"]
print("pineapple" in x)
```

False

# Conceptualizing Python in Google COLAB

```
x = ["apple", "banana"]
print("pineapple" not in x)

True
```

## Python Identity Operator (is)

Identity operators compare the memory locations of two objects. There are two Identity operators explained below –

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here <b>is</b> results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here <b>is not</b> results in 1 if id(x) is not equal to id(y).

In the following program, ‘x’ and ‘y’ are two distinct list objects while ‘x’ and ‘z’ are two distinct references pointing to the same list object. Hence all the three x, y and z have the same content and == operator used on them evaluates to ‘True’. While the ‘is’ operator used with the operands ‘x’ and ‘y’ evaluates to ‘False’. Hence == operator checks the content of the two operands passed to it while ‘is’ operator checks the objects referenced by the object variables.

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is z)
print(x is y)
print(x == z)
print(x == z)

True
False
True
True
```

# Conceptualizing Python in Google COLAB

---

Note: `is` compares object references where as `==` compares object content.

## Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

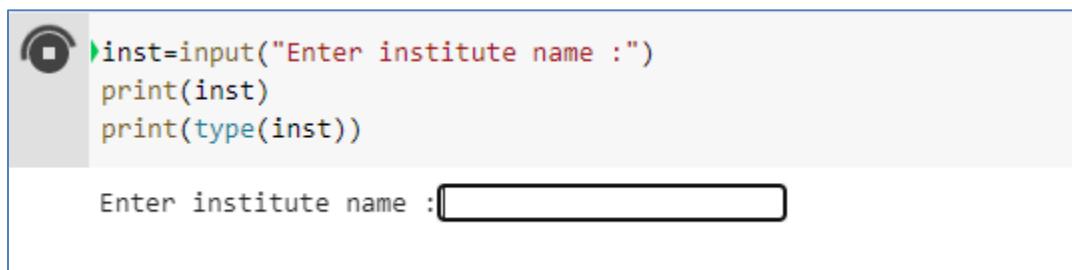
Sr.No.	Operator & Description
1	<code>**</code> Exponentiation (raise to the power)
2	<code>~ + -</code> Complement, unary plus and minus (method names for the last two are <code>+@</code> and <code>-@</code> )
3	<code>* / % //</code> Multiply, divide, modulo and floor division
4	<code>+ -</code> Addition and subtraction
5	<code>&gt;&gt; &lt;&lt;</code> Right and left bitwise shift
6	<code>&amp;</code> Bitwise 'AND'
7	<code>^  </code> Bitwise exclusive 'OR' and regular 'OR'

# Conceptualizing Python in Google COLAB

8	<b>&lt;= &lt; &gt; &gt;=</b> Comparison operators
9	<b>&lt;&gt; == !=</b> Equality operators
10	<b>= %= /= /= -= += *= **=</b> Assignment operators
11	<b>is is not</b> Identity operators
12	<b>in not in</b> Membership operators
13	<b>not or and</b> Logical operators

## Accepting Input from User

*input()* function is used for accepting input from the keyboard which returns a string value corresponding to the input entered by the user as demonstrated in the following figure:



The screenshot shows a Google Colab cell. The code is:

```
inst=input("Enter institute name :")
print(inst)
print(type(inst))
```

The output shows the prompt "Enter institute name :" followed by an empty input field.

# Conceptualizing Python in Google COLAB

```
▶ inst=input("Enter institute name :")
print(inst)
print(type(inst))
```

```
Enter institute name :SIBER
SIBER
<class 'str'>
```

For converting the input to the required data type, use one of the type conversion functions, int(), float(), bool() etc. In the following program, ‘age’ is accepted from the user and is converted into ‘int’.

```
▶ age=int(input("Enter age :"))
print(age)
print(type(age))
```

```
Enter age :20
20
<class 'int'>
```

## String formatting (Format String %)

The ‘%’ operator is employed for formatting a set of variables enclosed in a ‘tuple’ together with a format string, which contains normal text together with ‘*argument specifiers*’ or ‘argument place holders’ corresponding to special symbols such as ‘%s’ and ‘%d’ as demonstrated in the following programs:

```
▶ course="MSc"
inst="SIBER"
print("I am pursuing %s at CSIBER" % course)
```

```
I am pursuing MSc at CSIBER
```

```
▶ course="MSc"
inst="SIBER"
print("I am pursuing %s at %s" % (course,inst))
```

```
I am pursuing MSc at SIBER
```

# Conceptualizing Python in Google COLAB

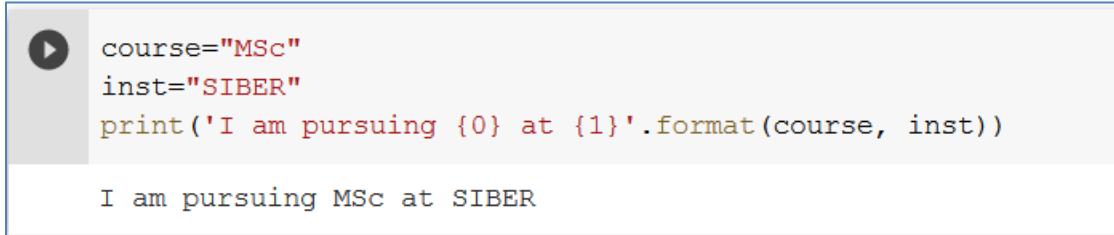
---

The different format codes supported by Python are enumerated in the following table:

‘d’ for integers
‘f’ for floating-point numbers
‘b’ for binary numbers
‘o’ for octal numbers
‘x’ for octal hexadecimal numbers
‘s’ for string
‘e’ for floating-point in an exponent format

## Formatting Output Using format()

The `format()` method was introduced in Python3 and contains the place holders enclosed in a pair of curly braces. There is one-to-one correspondence between place holders and parameters passed to `format()` method as demonstrated in the following program:



A screenshot of a Jupyter Notebook cell. On the left, there is a play button icon. The code in the cell is:

```
course="MSc"
inst="SIBER"
print('I am pursuing {0} at {1}'.format(course, inst))
```

The output of the cell is:

```
I am pursuing MSc at SIBER
```

## String Templates – Using F-Strings

F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting. It is also referred to as '**Literal String Interpolation**'. To create an F-String prefix the string with character '**F**'. The variables are part of the string and are enclosed in a pair of curly braces as demonstrated in the following programs:

# Conceptualizing Python in Google COLAB

```
▶ course="MSc"  
    inst="SIBER"  
    print(f"I am pursuing {course} at {inst}.")
```

I am pursuing MSc at SIBER.

```
▶ course="MSc"  
    inst="SIBER"  
    print("I am pursuing {course} at {inst}.")
```

I am pursuing {course} at {inst}.

```
▶ a = 5  
    b = 10  
    print(f"He said his age is {2 * (a + b)}.")
```

He said his age is 30.

## Generating Output in the Same Line

By default print() function generates output in a separate line as shown in the following program:

```
▶ print("Python is interpreted ")  
    print("and object-oriented")
```

Python is interpreted  
and object-oriented

To produce the output on the same line, use second named parameter in print() function specifying the line terminator which by default is new line character, '\n'.

The above program is re-written to produce the output in the same line.

```
▶ print("Python is interpreted ",end="")  
    print("and object-oriented")
```

Python is interpreted and object-oriented

# Conceptualizing Python in Google COLAB

---

## Chapter 4

### Lab Assignment on Basic Programs

#### Level – Basic

Program 1: To Check Whether the Given No is Positive, Negative or Zero

```
num=float(input("Enter a no. : "))
if (num > 0):
    print("Positive No.")
elif(num==0):
    print("Zero")
else:
    print("Negative No.")

Enter a no. : 10
Positive No.
```

```
num=float(input("Enter a no. : "))
if (num > 0):
    print("Positive No.")
elif(num==0):
    print("Zero")
else:
    print("Negative No.")

Enter a no. : -20
Negative No.
```

# Conceptualizing Python in Google COLAB

```
▶ num=float(input("Enter a no. : "))
  if (num > 0):
    print("Positive No.")
  elif(num==0):
    print("Zero")
  else:
    print("Negative No.")
```

```
Enter a no. : 0
Zero
```

## Using Nested if

```
▶ num=float(input("Enter a no. : "))
  if (num >= 0):
    if (num > 0):
      print("Positive No.")
    else:
      print("Zero")
  else:
    print("Negative No.")
```

```
Enter a no. : 10
Positive No.
```

```
▶ num=float(input("Enter a no. : "))
  if (num >= 0):
    if (num > 0):
      print("Positive No.")
    else:
      print("Zero")
  else:
    print("Negative No.")
```

```
Enter a no. : -10
Negative No.
```

# Conceptualizing Python in Google COLAB

```
▶ num=float(input("Enter a no. : "))
if (num >= 0):
    if (num > 0):
        print("Positive No.")
    else:
        print("Zero")
else:
    print("Negative No.")

Enter a no. : 0
Zero
```

## Program 2: To Check Whether the Given no. is Even or Odd

```
▶ num=int(input("Enter a no. : "))
if (num % 2) == 0:
    print("{0} is Even".format(num))
else:
    print("{0} is Odd".format(num))

Enter a no. : 8
8 is Even
```

```
▶ num=int(input("Enter a no. : "))
if (num % 2) == 0:
    print("{0} is Even".format(num))
else:
    print("{0} is Odd".format(num))

Enter a no. : 5
5 is Odd
```

# Conceptualizing Python in Google COLAB

## Program 3: To Find Largest Among Two No.s

```
▶ num1=int(input("Enter first no. : "))
  num2=int(input("Enter second no. : "))
  if (num1>num2):
    print("{0} is largest".format(num1))
  else:
    print("{0} is largest".format(num2))
```

```
Enter first no. : 10
Enter second no. : 20
20 is largest
```

```
▶ num1=int(input("Enter first no. : "))
  num2=int(input("Enter second no. : "))
  if (num1>num2):
    print("{0} is largest".format(num1))
  else:
    print("{0} is largest".format(num2))
```

```
Enter first no. : 50
Enter second no. : 10
50 is largest
```

## To Find Largest of Three No.s

```
▶ num1=int(input("Enter first no. : "))
  num2=int(input("Enter second no. : "))
  num3=int(input("Enter third no. : "))
  if (num1>num2 and num1>num3):
    print("{0} is largest".format(num1))
  elif(num2>num1 and num2>num3):
    print("{0} is largest".format(num2))
  else:
    print("{0} is largest".format(num3))
```

```
Enter first no. : 10
Enter second no. : 20
Enter third no. : 30
30 is largest
```

# Conceptualizing Python in Google COLAB

```
▶ num1=int(input("Enter first no. : "))
  num2=int(input("Enter second no. : "))
  num3=int(input("Enter third no. : "))
  if (num1>num2 and num1>num3):
    print("{0} is largest".format(num1))
  elif(num2>num1 and num2>num3):
    print("{0} is largest".format(num2))
  else:
    print("{0} is largest".format(num3))
```

```
↳ Enter first no. : 20
  Enter second no. : 10
  Enter third no. : 4
  20 is largest
```

```
▶ num1=int(input("Enter first no. : "))
  num2=int(input("Enter second no. : "))
  num3=int(input("Enter third no. : "))
  if (num1>num2 and num1>num3):
    print("{0} is largest".format(num1))
  elif(num2>num1 and num2>num3):
    print("{0} is largest".format(num2))
  else:
    print("{0} is largest".format(num3))
```

```
↳ Enter first no. : 10
  Enter second no. : 30
  Enter third no. : 20
  30 is largest
```

# Conceptualizing Python in Google COLAB

## Using Nested if

```
▶ num1=int(input("Enter first no. : "))
num2=int(input("Enter second no. : "))
num3=int(input("Enter third no. : "))
if (num1>num2):
    if(num1>num3):
        print("{0} is largest".format(num1))
    else:
        print("{0} is largest".format(num3))
else:
    if(num2>num3):
        print("{0} is largest".format(num2))
    else:
        print("{0} is largest".format(num3))
```

```
Enter first no. : 10
Enter second no. : 20
Enter third no. : 30
30 is largest
```

```
▶ num1=int(input("Enter first no. : "))
num2=int(input("Enter second no. : "))
num3=int(input("Enter third no. : "))
if (num1>num2):
    if(num1>num3):
        print("{0} is largest".format(num1))
    else:
        print("{0} is largest".format(num3))
else:
    if(num2>num3):
        print("{0} is largest".format(num2))
    else:
        print("{0} is largest".format(num3))
```

```
⇨ Enter first no. : 10
Enter second no. : 30
Enter third no. : 20
30 is largest
```

## Conceptualizing Python in Google COLAB

```
▶ num1=int(input("Enter first no. : "))
  num2=int(input("Enter second no. : "))
  num3=int(input("Enter third no. : "))
  if (num1>num2):
    if(num1>num3):
      print("{0} is largest".format(num1))
    else:
      print("{0} is largest".format(num3))
  else:
    if(num2>num3):
      print("{0} is largest".format(num2))
    else:
      print("{0} is largest".format(num3))
```

```
↳ Enter first no. : 30
  Enter second no. : 20
  Enter third no. : 10
  30 is largest
```

### Program 4: To Check Whether the Given No. is Prime or Not.

```
▶ num=int(input("Enter any number :"))
  flag=False
  if (num > 1):
    for i in range(2,num):
      if (num % i)==0:
        flag=True
        break
    if flag:
      print(num, " is not a Prime No.")
    else:
      print(num, " is a Prime No.")
```

```
↳ Enter any number :17
  17  is a Prime No.
```

# Conceptualizing Python in Google COLAB

```
▶ num=int(input("Enter any number :"))
  flag=False
  if (num > 1):
    for i in range(2,num):
      if (num % i)==0:
        flag=True
        break
    if flag:
      print(num, " is not a Prime No.")
    else:
      print(num, " is a Prime No.")
```

```
Enter any number :10
10  is not a Prime No.
```

## Program 5: To Compute the Factorial of a Given No.

```
▶ n=int(input("Enter ay no."))
  fact=1
  if ( n < 0):
    print("Enter Positive no...")
  elif ( n < 0):
    print("Factorial of 0 is 1")
  else:
    for i in range (1,(n+1)):
      fact*=i
    print("Factorial of %d is %d " % (n,fact))
```

```
Enter ay no.5
Factorial of 5 is 120
```

# Conceptualizing Python in Google COLAB

```
n=int(input("Enter ay no."))
fact=1
if ( n < 0):
    print("Enter Positive no...")
elif ( n < 0):
    print("Factorial of 0 is 1")
else:
    for i in range (1,(n+1)):
        fact*=i
    print("Factorial of %d is %d" % (n,fact))
```

Enter ay no.-4  
Enter Positive no...

```
n=int(input("Enter ay no."))
fact=1
if ( n < 0):
    print("Enter Positive no...")
elif ( n < 0):
    print("Factorial of 0 is 1")
else:
    for i in range (1,(n+1)):
        fact*=i
    print("Factorial of %d is %d" % (n,fact))
```

Enter ay no.0  
Factorial of 0 is 1

## Program 6: Python Program to display Multiplication Table

```
n=int(input("Enter any no."))
print("Multiplication Table for %d" % n)
for i in range(1,11):
    print("%2d x %d = %d" % (n,i,(n*i)))
```

Enter any no.5  
Multiplication Table for 5  
5 x 1 = 5  
5 x 2 = 10  
5 x 3 = 15  
5 x 4 = 20  
5 x 5 = 25  
5 x 6 = 30  
5 x 7 = 35  
5 x 8 = 40  
5 x 9 = 45  
5 x 10 = 50

# Conceptualizing Python in Google COLAB

## Program 7: Python Program to Print Fibonacci Sequence

```
▶ terms=int(input("Enter no. of terms"))
print("%d Term(s) of Fibonacii Sequence" % terms)
n1=0
n2=1
count=0
if (terms < 0):
    print("Enter positive terms")
elif (terms==1):
    print(0)
else:
    print([0])
    while(count < terms-1):
        term=n1+n2
        print(term)
        n1=n2
        n2=term
        count+=1

▶ Enter no. of terms7
7 Term(s) of Fibonacii Sequence
0
1
2
3
5
8
13
```

## Program 8: Python Program to Check Whether the Given No. is Armstrong No. or Not

```
▶ n=int(input("Enter any no."))
sum=0
temp=n
if (n<0):
    print("Enter positive no.")
else:
    while(temp > 0):
        rem=temp % 10
        sum+=rem*rem*rem
        temp // 10
    if (sum==n):
        print("%d is an Armstrong No." % n )
    else:
        print("%d is not an Armstrong No." % n )

▶ Enter any no.407
407 is an Armstrong No.
```

## Conceptualizing Python in Google COLAB

```
n=int(input("Enter any no."))
sum=0
temp=n
if (n<0):
    print("Enter positive no.")
else:
    while(temp > 0):
        rem=temp % 10
        sum+=rem*rem*rem
        temp //=
    if (sum==n):
        print("%d is an Armstrong No." % n )
    else:
        print("%d is not an Armstrong No." % n )
```

```
Enter any no.123
123 is not an Armstrong No.
```

### Program 9: Python Program to Reverse the Given No.

```
n=int(input("Enter any no."))
temp=n
rev=0
if (n < 0):
    print("Enter positive No.")
else:
    while (temp > 0):
        rem=temp % 10
        rev=rev*10+rem
        temp=temp//10
print("Reverse of the No. %d is %d" % (n,rev))
```

```
Enter any no.123456
Reverse of the No. 123456 is 654321
```

# Conceptualizing Python in Google COLAB

---

Program 10:

```
z = "xyz"  
j = 'j'  
if j in z:  
    print(j, end=" ")
```

*What will be the output of this statement?*

The screenshot shows a code cell in Google Colab. The code is identical to the one above, but the output is displayed in a separate box below the cell. The output shows the character 'x'.

```
[40] z = "xyz"  
      j = "j"  
      if j in z:  
          print(j, end=" ")  
  
x
```

Program 11

*What is output on execution of the following program in Python?*

```
x = 'pqrs'  
  
for i in range(len(x)):  
    x[i].upper()  
  
print (x)
```

# Conceptualizing Python in Google COLAB

---

```
[42] x = 'pqrs'
    for i in range(len(x)):
        x[i].upper()
    print (x)

pqrs

▶ x = 'pqrs'
for i in range(len(x)):
    print(x[i].upper(), end='')

PQRS
```

## Program 12

*What happens when '2' == 2 is executed?*

```
▶ print('2' == 2)

False
```

# Conceptualizing Python in Google COLAB

---

## Chapter 5

### Lab Assignment on Python Functions

#### Level – Basic

A function is a block of organized, reusable code which can be used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusability.

### Defining a Function

The rules for defining a function in Python are enumerated below:

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

The syntax for defining a function in Python is shown below:

#### Syntax:

```
def <function_name>(<parameter-list>):
    “function_docstring”
    function_suite
    return <expression>
```

# Conceptualizing Python in Google COLAB

---

## Example:

In the following example, printme() is a function which accepts a single string parameter and prints the same.

```
▶ def printme(str):
    "Function Demo"
    print(str)
    return

printme("Welcome to Python Programming")

Welcome to Python Programming
```

## Call By Value Vs. Call By Reference

Since everything in python is an object, the parameters passed to a function are object references which can be manipulated within the function body as demonstrated in the following program. In the following program, the list 'l' is passed to change() method which appends another list to the list passed as the argument.

```
▶ def change(list):
    "Call By Reference"
    list.append([1,2,3,4])
    return

l=[10,20,30,40]
print("List Before Calling change function")
print(l)
change(l)
print("List After Calling change function")
print(l)

List Before Calling change function
[10, 20, 30, 40]
List After Calling change function
[10, 20, 30, 40, [1, 2, 3, 4]]
```

# Conceptualizing Python in Google COLAB

---

## Call By Value

In the following program, ‘**list**’ is a variable local to **change()** method which is initialized and printed within the **change()** method. The list passed to **change()** method is not used in the function.

```
def change(list):
    "Call By Reference"
    list=[1,2,3,4]
    print("List inside Function ")
    print(list)
    return

list=[10,20,30,40]
change(list)
print("List outside function")
print(list)

List inside Function
[1, 2, 3, 4]
List outside function
[10, 20, 30, 40]
```

## Function Arguments

A function can be invoked by using the following types as formal arguments –

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

## Function with Required Arguments

Required arguments must be passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. In the following program, the **add()** function takes two required arguments, **num1** and **num2**. All the required arguments must be passed to the function in correct order as shown in the following program:

# Conceptualizing Python in Google COLAB

```
def add(num1,num2):  
    sum=num1+num2  
    return sum  
  
n1=int(input("Enter first no : "))  
n2=int(input("Enter second no : "))  
sum=add(n1,n2)  
print("Sum of %d and %d is %d" %(n1,n2,sum))
```

```
Enter first no : 10  
Enter second no : 20  
Sum of 10 and 20 is 30
```

## Function with Keyword Arguments or Named Arguments

In contrast to the required arguments, the keyword or named arguments need not maintain their position or order in the function call which enables to skip certain arguments containing default values in the function invocation. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. The above program is re-written below using named arguments. In the function definition, the parameter ‘num1’ occurs before the parameter ‘num2’. On the other hand, in the function invocation the parameter ‘num2’ occurs before the parameter ‘num1’ as shown in the following program:

```
def add(num1,num2):  
    sum=num1+num2  
    return sum  
  
n1=int(input("Enter first no : "))  
n2=int(input("Enter second no : "))  
sum=add(num2=n2,num1=n1)  
print("Sum of %d and %d is %d" %(n1,n2,sum))
```

```
Enter first no : 100  
Enter second no : 200  
Sum of 100 and 200 is 300
```

# Conceptualizing Python in Google COLAB

## Named Arguments

The following program demonstrates the use of named arguments in Python. When the function is invoked with different required and named parameters, the values of ‘a’, ‘b’ and ‘c’ are displayed in the output.

```
▶ def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c = 24)
func(c = 50, a = 100)

⇨ a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

## Function with Default Arguments

During the function definition certain default values can be assigned to the function parameters which can be employed if the corresponding parameter is not specified in the function invocation. In the following example, the above program is re-written where the add() function assigns the default values 10 and 20 to its parameters ‘num1’ and ‘num2’, respectively. If the add() function is invoked without one or both the parameters, then the default values will be utilized as demonstrated in the following program:

# Conceptualizing Python in Google COLAB

```
def add(num1=10,num2=20):
    sum=num1+num2
    return sum

sum=add()
print("Sum of %d and %d is %d" % (10,20,sum))

n1=int(input("Enter first no : "))
sum=add(n1)
print("Sum of %d and %d is %d" % (n1,20,sum))

n1=int(input("Enter first no : "))
n2=int(input("Enter second no : "))
sum=add(n1,n2)
print("Sum of %d and %d is %d" % (n1,n2,sum))

Sum of 10 and 20 is 30
Enter first no : 100
Sum of 100 and 20 is 120
Enter first no : 100
Enter second no : 200
Sum of 100 and 200 is 300
```

## Function with Variable No. of Arguments

Python supports variable no. of arguments which can be utilized if the no. of parameters to be passed to the function are not known at design time. An asterisk (\*) is placed before the variable name that holds the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. In the following example, terms is a variable no. of argument. The add() function must be invoked with minimum two required arguments.

# Conceptualizing Python in Google COLAB

```
def add(num1,num2,*terms):
    sum=num1+num2
    for x in terms:
        sum+=x
    return sum

sum=add(10,20)
print("Sum is %d" % (sum))

sum=add(10,20,30,40)
print("Sum is %d" % (sum))

sum=add(10,20,30,40,50,60)
print("Sum is %d" % (sum))

→ Sum is 30
Sum is 100
Sum is 210
```

## Anonymous or Lambda Functions

In Python, an anonymous function is a function that is defined without a name. While normal functions are defined using the ‘**def**’ keyword in Python, anonymous functions are defined using the **lambda** keyword. A lambda function can take any number of arguments, but can only have one expression. The syntax for defining a lambda function is shown below:

### Syntax:

*lambda <argument\_list>:expression*

### Rules in Defining Lambda Functions :

The following rules are observed in designing lambda functions:

- ‘lambda’ keyword is used in place of ‘def’ keyword for defining lambda functions.
- ‘lambda’ functions are anonymous which means the functions are defined without name.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

# Conceptualizing Python in Google COLAB

---

- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

The following examples demonstrate lambda functions for computing the sum of two no.s, doubling the no. and finding the maximum of two no.s passed to a function, respectively.

```
▶ sum=lambda num1,num2:num1+num2  
      print(sum(10,20))
```

30

```
▶ double=lambda num:num * 2  
      print("Double of %d is %d" % (10,double(10)))
```

Double of 10 is 20

```
▶ max=lambda x,y:x if x>y else y  
      print("Maximum of 100 and 200 is : ",max(100,200))
```

Maximum of 100 and 200 is : 200

## Examples:

Write a lambda function to increment the value by 10.

```
▶ x = lambda a : a + 10  
      print(x(5))
```

15

Write a lambda function to multiply two numbers.

```
▶ product = lambda a,b : a * b  
      print(product(4,5))
```

20

# Conceptualizing Python in Google COLAB

---

Write a lambda function to print a string



```
(lambda str : print(str)) ("Hello lambda Function")
```

```
Hello lambda Function
```

## Lambda with Default Arguments

A lambda function can take default arguments. In the following example, a lambda function is defined with three parameters x, y and z which are assigned default values 10, 20 and 30, respectively.



```
sum=lambda x=10,y=20, z=30 : x+y+z
print("Sum of 10, 20 and 30 : ",sum())
print("Sum of 100, 20 and 30 : ",sum(100))
print("Sum of 100, 200 and 30 : ",sum(100,200))
print("Sum of 100, 200 and 300 : ",sum(100,200,300))

Sum of 10, 20 and 30 : 60
Sum of 100, 20 and 30 : 150
Sum of 100, 200 and 30 : 330
Sum of 100, 200 and 300 : 600
```

## Applications of Lambda Functions

lambda function can be used anywhere where a nameless function is required for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

### filter() Function

The filter() function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True. Here is an example use of filter() function to filter out only even numbers from a list.

# Conceptualizing Python in Google COLAB

```
list1=[1,2,3,4,5,6,7,8,9,10]
even_list=__builtins__.list(filter(lambda x:(x%2==0),list1))
print("Origianal List : ",list1)
print("Even list      : ",even_list)
```

```
Origianal List : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Even list      : [2, 4, 6, 8, 10]
```

In the following example, filter() function is employed to filter out all even and odd no.s from a list, list1.

```
list1=[1,2,3,4,5,6,7,8,9,10]
even_list=__builtins__.list(filter(lambda x:(x%2==0),list1))
odd_list=__builtins__.list(filter(lambda x:(x%2!=0),list1))
print("Origianal List : ",list1)
print("Even list      : ",even_list)
print("Odd list       : ",odd_list)
```

```
Origianal List : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Even list      : [2, 4, 6, 8, 10]
Odd list       : [1, 3, 5, 7, 9]
```

## map() Function

The map() function in Python takes in a function and a list. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item. Here is an example use of map() function to double all the items in a list.

```
list1=[1,2,3,4,5,6,7,8,9,10]
doubled_list=__builtins__.list(map(lambda x:2*x,list1))
print("Origianal List   : ",list1)
print("Doubled list    : ",doubled_list)
```

```
Origianal List   : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Doubled list    : [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Map each element of a list o its square using lambda function

# Conceptualizing Python in Google COLAB

```
numbers=[1,2,3,4,5]
squares=list(map(lambda x:x*x,numbers))
print("original No.s : ",numbers)
print("Square No.s   : ",squares)

original No.s :  [1, 2, 3, 4, 5]
Square No.s   :  [1, 4, 9, 16, 25]
```

## Higher Order Functions

In Python, a function can return a function and a function can take functions as arguments such functions are referred to as '**Higher Order Functions**'. In the following example, myfun() is a higher order function which takes a single parameter and returns a lambda function taking one parameter. Thus, myfun() can be invoked with different values for creating different functions for doubling, tripling no.s etc. as demonstrated in the following program:

```
def myfun(n):
    return lambda a:a*n

mydoubler=myfun(2)
print(mydoubler(10))

mytrippler=myfun(3)
print(mytrippler(10))

20
30
```

## Example:

In the following example, arithmetic() function takes three parameters. The first parameter is a function which accepts exactly two arguments. The next two parameters are the parameters to the function passed as a first argument to arithmetic() function. add(), subtract(), multiply() and divide() are the functions which take exactly two parameters and hence can be passed as first parameter to arithmetic() function for performing different arithmetic operations on two no.s.

# Conceptualizing Python in Google COLAB

```
def arithmetic( funct, num1, num2 ):
    return funct( num1, num2 )
def add( num1, num2 ):
    return num1+num2
def subtract( num1, num2 ):
    return num1-num2
def multiply( num1, num2 ):
    return num1*num2
def divide( num1, num2 ):
    return num1/num2

print("Sum of 10 and 20      : ",arithmetic(add,10,20))
print("Difference of 10 and 20 : ",arithmetic(subtract,10,20))
print("Multiplication of 10 and 20 : ",arithmetic(multiply,10,20))
print("Division of 10 and 20      : ",arithmetic(divide,10,20))
```

```
↳ Sum of 10 and 20      : 30
    Difference of 10 and 20 : -10
    Multiplication of 10 and 20 : 200
    Division of 10 and 20      : 0.5
```

## Higher Order Function with Variable No. of Arguments

The following program demonstrates a higher order function, arithmetic() which accepts two parameters, the first parameter is a function which accepts variable no. of argument as its only parameter and the second parameter is a variable no. of arguments to a function. The arithmetic function is invoked by passing add() function as its first parameter and variable no. of arguments as shown below:

```
def arithmetic(func, *num):
    return func(*num)

def add(*num):
    sum=0
    for ele in num:
        sum+=ele
    print (sum)

arithmetic(add,10,20)
arithmetic(add,10,20,30)
arithmetic(add,10,20,30,40)
```

```
30
60
100
```

# Conceptualizing Python in Google COLAB

---

## Chapter 6

### Lab Assignment on Advanced Concepts in Python - I

(Covers Iterators, Closures, Decorators and Generators)

Level – Intermediate

#### Using Iterator

Iterators are elegantly implemented within for loops, comprehensions, generators etc. Iterator in Python is an object which can be iterated upon and returns data, one element at a time. An iterator object implements two special methods `__iter__()` and `__next__()` which are collectively referred to as '**Iterator Protocol**'.

#### Iterables

An object is said to be iterator, if it supports `__iter__()` method which returns an iterator. Most of the in-built classes in Python such as list, tuple, set, string are all iterables.

#### `__next__()` Method of Iterator

An iterator supports `__next__()` method to manually iterate through the items of an iterator. The `__next__()` method throws ‘StopIteration’ exception when end of the iteration is reached and no more data is found as shown in the following programs:

```
mylist=[1,2]
myiter=mylist.__iter__()
print(myiter.__next__())
print(myiter.__next__())
```

```
1
2
```

# Conceptualizing Python in Google COLAB

```
▶ mylist=[1,2]
myiter=mylist.__iter__()
print(myiter.__next__())
print(myiter.__next__())
print(myiter.__next__())

1
2
-----
StopIteration                                     Traceback (most recent call last)
<ipython-input-11-81d21eefc3b6> in <module>()
      3 print(myiter.__next__())
      4 print(myiter.__next__())
----> 5 print(myiter.__next__())

StopIteration:
```

[SEARCH STACK OVERFLOW](#)

## Iterating through the list Using for Loop

The most elegant way to iterate through the iterator is using for loop which successively invokes `__next__()` method on the iterator to traverse through the elements and processes ‘`StopIteration`’ exception when all the elements of the iterator are exhausted. The following program demonstrates iterating through the list using for loop.

```
▶ mylist=[1,2]
for ele in mylist:
    print(ele)

1
2
```

## Implementation of for loop - Handling StopIteration Exception

The following program demonstrates the actual implementation of the for loop for iterating through the list. The infinite while loop is employed for iterating through the elements which encapsulates try..except block for exception handling. ‘break’ statement is used in ‘except’ block

# Conceptualizing Python in Google COLAB

---

to break from the loop when ‘**StopIteration**’ exception is thrown and no more elements exist in the iterator.

```
mylist=[1,2,3,4,5]
myiter=mylist.__iter__()
while True:
    try:
        print(myiter.__next__())
    except StopIteration:
        break

1
2
3
4
5
```

## Nested Functions in Python

In Python a function can be nested inside another function to create a nested function structure as shown below. As shown in the program, nested functions can access variables of the enclosing scope which are non-local members of the function. In the following program, greet() is a function which is nested inside greetings() function and exists within the scope of greetings() function. The greet() function has access to the parameter ‘nm’ of enclosing function as shown:

```
def greetings(nm):
    def greet():
        print("Hello "+nm)
    greet()

greetings("Student")
greetings("MSc")

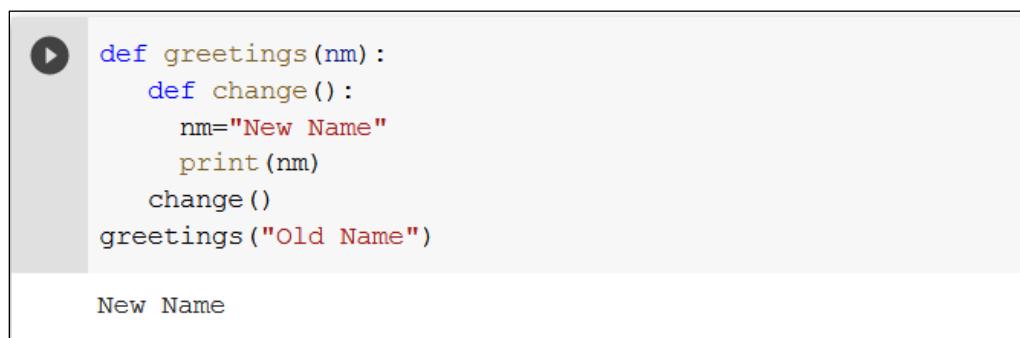
Hello Student
Hello MSc
```

This technique by which some data (‘Student’ or ‘MSc’ in this case) gets attached to the code is called closure in Python. This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

# Conceptualizing Python in Google COLAB

---

Also, the nested function can modify the non-local variables as demonstrated in the following program:



```
def greetings(nm):
    def change():
        nm="New Name"
        print(nm)
    change()
greetings("Old Name")
```

New Name

As seen in the above program, the nested printer() function was able to access the non-local ‘nm’ variable of the enclosing function.

## Closure Functions

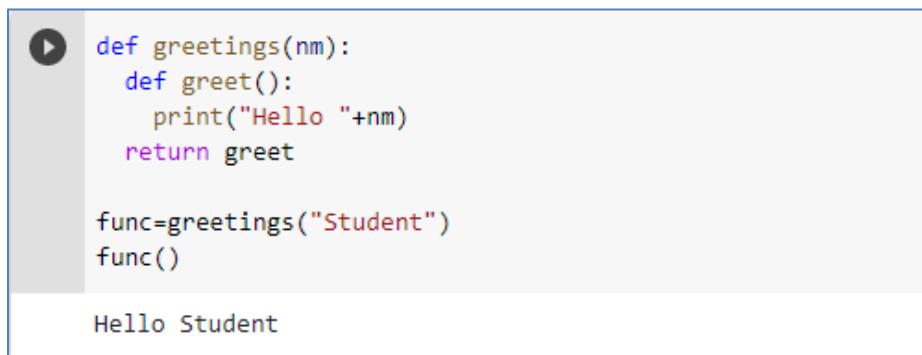
We have a closure in Python when a nested function references a value in its enclosing scope

### Criteria for Closures

The criteria that must be met to create closure in Python are enumerated below:

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

In the case of a closure function, a function is nested within the scope of outer function and is returned by the outer function as shown in the following program. In the following program, the greetings() function is publicly visible which returns a nested ‘greet’ function.



```
def greetings(nm):
    def greet():
        print("Hello "+nm)
    return greet

func=greetings("Student")
func()
```

Hello Student

# Conceptualizing Python in Google COLAB

---

In the above program, the greetings() function was called with the string ‘Student’ and the returned function was bound to the variable ‘func’. On calling func(), the ‘nm’ was still remembered although we had already finished executing the greetings() function.

In Python everything is an object, including a function. Hence the function reference can be stored in a variable which can then be invoked any no. of times as demonstrated in the following program:

```
def greeting1():
    print("Hello from python")

greeting2=greeting1
greeting3=greeting1

greeting1()
greeting2()
greeting3()
```

```
Hello from python
Hello from python
Hello from python
```

In the above program, ‘greeting2’ and ‘greeting3’ both refer to the same function object.

In the following program, the ‘*greet*’ function reference returned by ‘*greetings*’ function is stored in a variable ‘*func*’ before deleting the ‘*greetings*’ function.

```
def greetings(nm):
    def greet():
        print("Hello "+nm)
        return greet

func=greetings("Student")
func()
del greetings
func()
```

```
Hello Student
Hello Student
```

Here, the returned function still works even when the original function was deleted.

# Conceptualizing Python in Google COLAB

```
▶ def greetings(nm):
    def greet():
        print("Hello "+nm)
    return greet

func=greetings("Student")
func()
del greetings
del greet

Hello Student
-----
NameError Traceback (most recent call last)
<ipython-input-3-3be216244024> in <module>()
      7 func()
      8 del greetings
----> 9 del greet

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

SEARCH STACK OVERFLOW

```
▶ def greetings(nm):
    def greet():
        print("Hello "+nm)
    return greet

func=greetings("Student")
func()
del greetings
del func
```

Hello Student

## Example of Closure Function:

In the following example, make\_multiplier\_of() is a function accepting argument ‘n’ which creates a multiplier() function with argument ‘m’ and multiplies the parameter passed to it with ‘n’.

# Conceptualizing Python in Google COLAB

```
def make_multiplier_of(n):
    def multiplier(m):
        return m*n
    return multiplier

multiplier2=make_multiplier_of(2)
print(multiplier2(10))

multiplier3=make_multiplier_of(3)
print(multiplier3(10))

multiplier4=make_multiplier_of(4)
print(multiplier4(10))

20
30
40
```

## Applications of Closures

Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

## Higher Order Functions

A function in Python can take other functions as arguments and return a function. Such functions are referred to as higher order functions. In the following program, change() function accepts two parameters, the first parameter is a name of the function to be invoked which accepts exactly one parameter and the second argument is the parameter to be passed to a function. inc() and dec() are functions which accept exactly one parameter and hence can be passed to change() function. inc() function increments the value of the argument passed to it ad dec() function decrements the value of the argument passed to it.

# Conceptualizing Python in Google COLAB

```
def inc(x):
    return x+1

def dec(x):
    return x-1

def change(func, x):
    return func(x)

y=change(inc,10)
print(y)

y=change(dec,10)
print(y)
```

```
11
9
```

The above program is re-written below where inc() and dec() functions now accept two parameters, the original no and the value by which the number should be incremented or decremented.

```
def inc(x,m):
    return x+m

def dec(x,m):
    return x-m

def change(func, x,y):
    return func(x,y)

y=change(inc,10,5)
print(y)

y=change(dec,10,5)
print(y)
```

```
15
5
```

# Conceptualizing Python in Google COLAB

---

In the following example, C\_called() is a higher order function which creates and returns a function Python\_returned().



```
def C_called():
    def Python_returned():
        print("Hello from Python")
        return Python_returned

    func=C_called()
    func()

Hello from Python
```

The function C\_called() can be directly invoked without assigning it to a variable as shown in the following program:



```
def C_called():
    def Python_returned():
        print("Hello from Python")
        return Python_returned

    C_called()()

Hello from Python
```

## Decorators in Python

A decorator in Python is a function which takes a function as its argument adds some functionality and returns it. This is also called '**‘Meta Programming’**' because a part of the program tries to modify another part of the program at compile time. In the following program, **decorated\_greetings()** is a decorator which displays a decorated start line before and after invoking the function passed to it as an argument.

# Conceptualizing Python in Google COLAB

```
def decorated_greetings(func):
    print("*****")
    func()
    print("*****")

def ordinary_greetings():
    print("Hello from python")

ordinary_greetings()
print("\n")
decorated_greetings(ordinary_greetings)
```

```
Hello from python
```

```
*****
Hello from python
*****
```

In the following program, the function *decorated\_greetings()* is bound to a variable ‘decorated’ which passes a function *ordinary\_greetings()* for decoration. Finally, the decorated function, *decorated()* is invoked. *ordinary\_greetings()* function was decorated and the returned function was given the name ‘*decorated*’.

```
def decorated_greetings(func):
    def inner():
        print("*****")
        func()
        print("*****")
    return inner

def ordinary_greetings():
    print("Hello from python")

decorated=decorated_greetings(ordinary_greetings)
decorated()
```

```
*****
Hello from python
*****
```

# Conceptualizing Python in Google COLAB

---

## Callable

Functions and methods are called callable as they can be called.

In fact, any object which implements the special `__call__()` method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

```
def greetings(nm):
    def greet():
        print("Hello "+nm)
    return greet

greetings.__call__("SIBER")
<function __main__.greetings.<locals>.greet>
```

## Using Annotation

This is a common construct and for this reason, Python has a syntax to simplify this. The function to be decorated can be modified with the header which contains '@' symbol along with the name of the decorator function to which it is to be passed.

```
@decorator_func
def ordinary_func():
    //function suit
```

is equivalent to

```
decorator_func(ordinary_func)
```

In the following example, `ordinary_greetings()` function is passed to a decorator `decorated_greetings()`. Hence on invocation of `ordinary_greetings()` function automatically `decorated_greetings()` function is invoked by passing `ordinary_greetings()` function as its parameter.

# Conceptualizing Python in Google COLAB

```
def decorated_greetings(func):
    def inner():
        print("*****")
        func()
        print("*****")
    return inner

@decorated_greetings
def ordinary_greetings():
    print("Hello from python")

ordinary_greetings()
```

```
*****
Hello from python
*****
```

Decorators also employ closures. In the following example, outer\_func() is a function which is a decorator to test() function accepting two arguments. outer\_func() defines a nested function, inner\_func() which accepts two arguments and prints them. Since the nested function has access to the parameters of outer function, the parameters passed to a function func() are accessed by inner\_func() and the same are printed.

```
def outer_func(func):
    def inner_func(a,b):
        print(a)
        print(b)
    return inner_func

@outer_func
def test(a,b):
    print("Test")

test(10,20)
```

```
10
20
```

# Conceptualizing Python in Google COLAB

The no. of parameters passed to the function to be decorated should exactly match parameters of nested function.

```
def outer_func(func):
    def inner_func(a,b):
        print(a)
        print(b)
    return inner_func

@outer_func
def test(a,b):
    print("Test")

test(10,20,30)

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-54-c4939c451cc5> in <module>()
      10
      11
---> 12 test(10,20,30)

TypeError: inner_func() takes 2 positional arguments but 3 were given
```

## Application to Division

In the following example, `divide()` is a function which accepts two parameters, divides the first parameter by the second and returns the same.

```
def divide(a,b):  
    return a/b  
  
divide(10,5)
```

When the divide() function is invoked by passing 0 as second parameter, the exception ‘ZeroDivisionError’ is thrown as shown in the following figure:

# Conceptualizing Python in Google COLAB

```
▶ def divide(a,b):
...     return a/b
...
divide(10,0)

-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-30-55af3e76ae8e> in <module>()
      2     return a/b
      3
----> 4 divide(10,0)
      5

<ipython-input-30-55af3e76ae8e> in divide(a, b)
      1 def divide(a,b):
----> 2     return a/b
      3
      4 divide(10,0)
      5

ZeroDivisionError: division by zero
```

SEARCH STACK OVERFLOW

The problem can be addressed using a decorator. In the following program, smart\_divide() is a decorator to a divide() function which examines the second parameter before dividing the first parameter by the second. If the second parameter is zero, the message ‘Cannot Divide a by 0’ is printed and the function returns. If the second parameter is not zero, divide() function is invoked by passing the two parameters as demonstrated in the following programs:

```
▶ def smart_divide(func):
    def inner(a,b):
        print("Division of a by b")
        if (b==0):
            print("Cannot Divide a by 0")
            return
        return func(a,b)
    return inner

@smart_divide
def divide(a,b):
    print(a/b)

divide(20,10)

Division of a by b
2.0
```

# Conceptualizing Python in Google COLAB



```
def smart_divide(func):
    def inner(a,b):
        print("Division of a by b")
        if (b==0):
            print("Cannot Divide a by 0")
            return
        return func(a,b)
    return inner

@smart_divide
def divide(a,b):
    print(a/b)

divide(20,0)
```

```
Division of a by b
Cannot Divide a by 0
```

Note: Parameters of the nested inner() function inside the decorator is the same as the parameters of functions it decorates.

## Chapter 7

### Lab Assignment on Advanced Concepts in Python - II

Level – Intermediate

#### List Comprehension

List comprehension offers a shorter syntax for creating a new list from the values of an existing list based on the condition. The element from the specified list is selected if the condition evaluates to ‘True’, otherwise the element is discarded. The syntax for list comprehension is shown below:

Syntax:

# Conceptualizing Python in Google COLAB

---

***newlist = [expression for item in iterable if condition == True]***

The return value is a new list, leaving the old list unchanged.

## Code Without List Comprehension

The following program creates a new list based on list ‘fruits’ which contains all elements containing the letter ‘a’ in their names without employing list comprehension.

```
▶ fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  newlist = []
  for x in fruits:
    if "a" in x:
      newlist.append(x)
  print(newlist)

['apple', 'banana', 'mango']
```

## Code with List Comprehension

In the following example, the above program is re-written employing list comprehension which renders the code compact.

```
▶ fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  newlist = [x for x in fruits if "a" in x]
  print(newlist)

['apple', 'banana', 'mango']
```

## Example

Set all values in the new list to ‘mango’:

# Conceptualizing Python in Google COLAB

---

```
newlist = ['mango' for x in fruits]
```

The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome:

## Example

Return "orange" instead of "banana": (Replaces banana with orange)

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

The expression in the example above can be interpreted as :

**'Return the item if it is not banana, if it is banana return orange'.**

The condition is optional and can be omitted as shown below:

Without if statement:

```
newlist = [x for x in fruits]
```

The above code creates a new list by copying all elements of list 'fruits'.

## Example:

The following programs demonstrate list comprehension for selecting the elements in the given range employing a range() function and a condition for filtering the elements from the list.

```
▶ newlist = [x for x in range(10) if x < 5]
print(newlist)
[0, 1, 2, 3, 4]
```

# Conceptualizing Python in Google COLAB

```
n=10  
newlist = [x for x in range(100) if x < n]  
print(newlist)  
  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Modifying Output

The elements of the original list can be modified while generating a new list. The following program generates a new list which contains the square root of all elements in original list using list comprehension.

```
alist = [4, 16, 64, 256]  
out = [a**(1/2) for a in alist]  
print(out)  
  
[2.0, 4.0, 8.0, 16.0]
```

In the following program, a new list '**result1**' is generated based on the contents of '**result**' list which maps the elements '**p**' and '**f**' to '**Pass**' and '**Fail**', respectively.

```
result = ['p', 'p', 'f', 'f']  
result1 = ['Pass' if x=='p' else 'Fail' for x in result]  
print(result1)  
  
['Pass', 'Pass', 'Fail', 'Fail']
```

## Generator Function

A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function. In the following example, `gen_fun()` is a generator function which yields four values 1, 2, 3 and 4. Similar to iterables such as list, tuple, set etc., a generator function can be used in for loop for retrieving its elements which invokes `__next__()` method on generator function object in each iteration. Similar to iterables, the `StopIteration`

# Conceptualizing Python in Google COLAB

---

exception is thrown when `__next__()` method is invoked on a generator function and value is not yielded.

```
def gen_fun():
    yield 1
    yield 2
    yield 3
    yield 4
for ele in gen_fun():
    print(ele)

1
2
3
4
```

## Return vs. Yield

The return is a final statement of a function. It provides a way to send some value back. While returning, its local stack also gets flushed. And any new call will begin execution from the very first statement.

On the contrary, the yield preserves the state between subsequent function calls. It resumes execution from the point where it gave back the control to the caller, i.e., right after the last yield statement.

## Generator Object

Generator function returns a generator object which is iterable. The value yielded by generator function can be accessed using one of the following approaches:

### Method 1:

Using the generator object in ‘for in’ loop.

# Conceptualizing Python in Google COLAB

---

## Method 2:

Invoking `__next__()` method on generator object as demonstrated in the following program:

The screenshot shows a Jupyter Notebook cell. On the left, there is a play button icon. The code in the cell is as follows:

```
def gen_fun():
    yield 1
    yield 2
    yield 3
    yield 4
obj=gen_fun()
print(obj.__next__())
print(obj.__next__())
print(obj.__next__())
print(obj.__next__())
```

Below the code, the output is displayed in four lines:

```
1
2
3
4
```

An attempt to invoke `__next__()` method on generator object past yielded elements throws '**StopIteration**' exception as demonstrated in the following program:

# Conceptualizing Python in Google COLAB

```
▶ def gen_fun():
    yield 1
    yield 2
    yield 3
    yield 4
obj=gen_fun()
print(obj.__next__())
print(obj.__next__())
print(obj.__next__())
print(obj.__next__())
print(obj.__next__())

1
2
3
4
-----
StopIteration                                     Traceback (most recent call last)
<ipython-input-15-28dec8095e1d> in <module>()
      9 print(obj.__next__())
     10 print(obj.__next__())
--> 11 print(obj.__next__())

StopIteration:
```

SEARCH STACK OVERFLOW

## Generator Yielding Tuple

The yield statement can return more than one value at a time in a tuple as demonstrated in the following program. In the following example, `gen()` is a generator function which yields two tuples containing two elements each.

```
▶ def gen():
    x, y = 1, 2
    yield x, y
    x += 1
    yield x, y
g = gen()
print(next(g))
print(next(g))
try:
    print(next(g))
except StopIteration:
    print("Iteration finished")
```

```
(1, 2)
(2, 2)
Iteration finished
```

# Conceptualizing Python in Google COLAB

---

## Python generator expression

Generator expression is similar to a list comprehension. The difference is that a generator expression returns a generator, not a list.

A generator expression is created with round brackets. Creating a list comprehension in this case would be very inefficient because the example would occupy a lot of memory unnecessarily. Instead of this, we create a generator expression, which generates values lazily on demand.

### Example:

```
n = (e for e in range(50) if not e % 3)
```

```
i = 0
```

```
for e in n:
```

```
    print(e)
```

```
    i += 1
```

```
    if i > 100:
```

```
        raise StopIteration
```

```
n = (e for e in range(50) if not e % 3)
```

```
for e in n:
```

```
    print(e)
```

## Generator Expression

```
# Demonstrate Python Generator Expression
```

```
# Define the list
```

```
alist = [4, 16, 64, 256]
```

```
# Find square root using the Generator Function
```

```
out = (a***(1/2) for a in alist)
```

```
for e in out:
```

```
    print(e)
```

# Conceptualizing Python in Google COLAB

```
alist = [4, 16, 64, 256]
# Find square root using the list comprehension
out = [a**(1/2) for a in alist]
print(type(out))

alist = [4, 16, 64, 256]
# Find square root using the list comprehension
out = (a**(1/2) for a in alist)
print(type(out))

<class 'list'>
<class 'generator'>
```

```
# Generator next() Method Demo
alist = ['Python', 'Java', 'C', 'C++', 'CSharp']
def list_items():
    for item in alist:
        yield item
gen = list_items()
iter = 0
while iter < len(alist):
    print(next(gen))
    iter += 1
```

```
Python
Java
C
C++
CSharp
```

Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]
# For loop to reverse the string
for char in rev_str("hello"):
    print(char)

o
l
l
e
h
```

# Conceptualizing Python in Google COLAB

```
a = (x*x for x in range(10))
print(sum(a))
```

285

## Closure Functions

The following program demonstrates a closure function where the '*outer*' function returns an '*inner*' function.

```
def outer(num1):
    def inner():
        print(num1)
    inner()
outer(10)
```

10

In closures, the outer function does not have an access to the parameters of inner function. At attempt to access the parameters of inner function inside outer function generates '*NameError*'.

```
def outer(num1):
    print(num2)
    def inner(num2):
        print(num1)
    inner(20)
outer(10)
```

```
-----  
NameError                                 Traceback (most recent call last)
<ipython-input-2-287ec29d7544> in <module>()
      4     print(num1)
      5     inner(20)
----> 6 outer(10)
```

```
<ipython-input-2-287ec29d7544> in outer(num1)
      1 def outer(num1):
----> 2     print(num2)
      3     def inner(num2):
      4         print(num1)
      5     inner(20)
```

```
NameError: name 'num2' is not defined
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

However, the inner function can access the parameters of the outer function as demonstrated in the following program:

```
def outer(num1):
    def inner(num2):
        print(num1)
        print(num2)
    inner(20)
outer(10)
```

```
10
20
```

The following program demonstrates generator function yielding the elements of a list.

```
alist = ['Python', 'Java', 'C', 'C++', 'CSharp']
def list_items():
    for item in alist:
        yield item
gen = list_items()
iter = 0
while iter < len(alist):
    print(next(gen))
    iter += 1
```

```
Python
Java
C
C++
CSharp
```

The following program demonstrates reversing the string using a generator function.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]

# For loop to reverse the string
for char in rev_str("hello"):
    print(char,end='')
```

```
olleh
```

# Conceptualizing Python in Google COLAB

The following program creates a tuple which contains the square of the elements in the range 0 to 9. The aggregate function sum() is used for computing the sum of square of natural no.s in the range 0 to 9.

```
a = (x*x for x in range(10))
print(sum(a))
```

285

## Generator Pipeline

The different generator functions with distinct functionality can be pipelined together to generate a cumulative effect. In the following example, three generator functions are defined:

even\_filter → It accepts a list and yields only even no.s from the list.

multily\_by\_three → It accepts a list, multiplies each element of the list by 3 and yields.

convert\_to\_string → It accepts a list and yields the string representation of each element in the list.

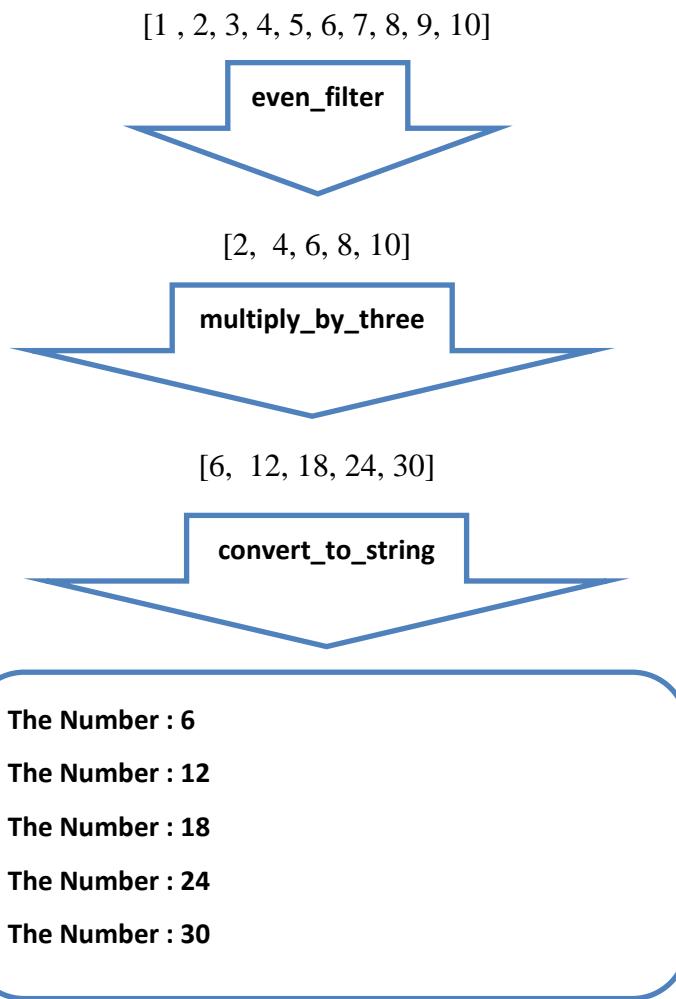
```
def even_filter(nums):
    for num in nums:
        if num % 2 == 0:
            yield num
def multiply_by_three(nums):
    for num in nums:
        yield num * 3
def convert_to_string(nums):
    for num in nums:
        yield 'The Number: %s' % num
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pipeline = convert_to_string(multiply_by_three(even_filter(nums)))
for num in pipeline:
    print(num)
```

```
▶ The Number: 6
The Number: 12
The Number: 18
The Number: 24
The Number: 30
```

# Conceptualizing Python in Google COLAB

---

In the main program, the list is defined which is passed as a parameter to a generator function, even\_filter to extract all even no.s from the list. The output of even\_filter generator function becomes input to multily\_by\_three generator function, which multiplies each element by 3 and passed as input to third generator function which displays each no. with the string ‘The Number: ‘ prepended to it. The entire generator pipeline is depicted in the following figure:



The following example demonstrates yet another instance of generator pipeline. The generator function `pass_filter()` accepts a list of marks and yields only those values for which marks is greater than equal to 40. The output of `pass_filter()` generator is passed as input to `add_bonus()` generator function which adds 10 bonus marks to each passed student. Next, ‘`for`’ loop iterates through the list and prints the elements.

# Conceptualizing Python in Google COLAB

```
def pass_filter(marks):
    for mark in marks:
        if mark >= 40:
            yield mark

def add_bonus(marks):
    for mark in marks:
        yield mark+10
marks = [35,40,30,55,60]
pipeline = add_bonus(pass_filter(marks))
for mark in pipeline:
    print(mark)

50
65
70
```

The entire generator pipeline is depicted in the following figure:



## Chapter 8

### Lab Assignment on Exception Handling in Python

Level – Intermediate

#### Exception

A typical program can contain one of the following types of errors:

- Logical Error
- Compiler Error
- Linker Error
- Runtime Error

Logical error can be trapped through exhaustive test cases. Compiler error is tracked by the compiler and can be fixed, so is the linker error. If the program contains the runtime error, then it will compile successfully. However, during the execution will generate an error. Such a runtime error is referred to as '***Exception***'.

An exception can be defined as an unusual condition in a program which the program cannot cope up with, resulting in the interruption in the flow of the program.

A program can employ one of the following approaches:

- anticipate and handle the exception in the program and take the corrective measures
- leave it to the system for processing

#### Standard or Pre-defined Exception in Python

List of standard exceptions is depicted in the following table:

# Conceptualizing Python in Google COLAB

---

Sr.No.	Exception Name & Description
1	<b>Exception</b> Base class for all exceptions
2	<b>StopIteration</b> Raised when the next() method of an iterator does not point to any object.
3	<b>SystemExit</b> Raised by the sys.exit() function.
4	<b>StandardError</b> Base class for all built-in exceptions except StopIteration and SystemExit.
5	<b>ArithmeticError</b> Base class for all errors that occur for numeric calculation.

6	<b>OverflowError</b> Raised when a calculation exceeds maximum limit for a numeric type.
7	<b>FloatingPointError</b> Raised when a floating point calculation fails.
8	<b>ZeroDivisionError</b> Raised when division or modulo by zero takes place for all numeric types.
9	<b>AssertionError</b> Raised in case of failure of the Assert statement.
10	<b>AttributeError</b> Raised in case of failure of attribute reference or assignment.

# Conceptualizing Python in Google COLAB

---

11	<b>EOFError</b> Raised when there is no input from either the <code>raw_input()</code> or <code>input()</code> function and the end of file is reached.
12	<b>ImportError</b> Raised when an import statement fails.
13	<b>KeyboardInterrupt</b> Raised when the user interrupts program execution, usually by pressing <code>Ctrl+c</code> .
14	<b>LookupError</b> Base class for all lookup errors.
15	<b>IndexError</b> Raised when an index is not found in a sequence.

16	<b>KeyError</b> Raised when the specified key is not found in the dictionary.
17	<b>NameError</b> Raised when an identifier is not found in the local or global namespace.
18	<b>UnboundLocalError</b> Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	<b>EnvironmentError</b> Base class for all exceptions that occur outside the Python environment.
20	<b>IOError</b> Raised when an input/ output operation fails, such as the <code>print</code> statement or the <code>open()</code> function when trying to open a file that does not exist.

# Conceptualizing Python in Google COLAB

---

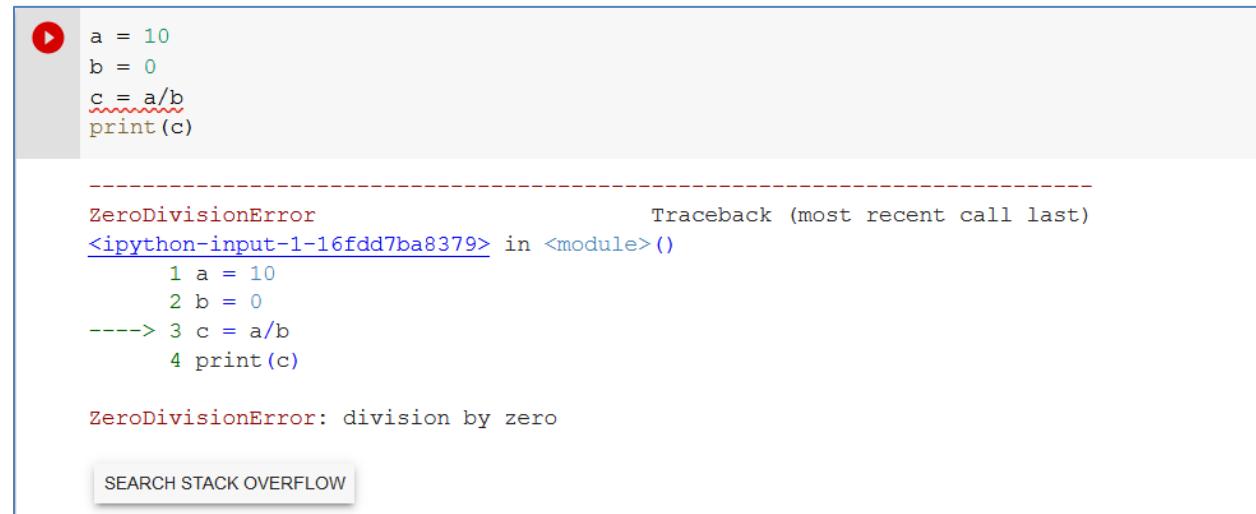
21	<b>IOError</b> Raised for operating system-related errors.
22	<b>SyntaxError</b> Raised when there is an error in Python syntax.
23	<b>IndentationError</b> Raised when indentation is not specified properly.
24	<b>SystemError</b> Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	<b>SystemExit</b> Raised when Python interpreter is quit by using the <code>sys.exit()</code> function. If not handled in the code, causes the interpreter to exit.
26	<b>TypeError</b> Raised when an operation or function is attempted that is invalid for the specified data type.
27	<b>ValueError</b> Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	<b>RuntimeError</b> Raised when a generated error does not fall into any category.
29	<b>NotImplementedError</b> Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

# Conceptualizing Python in Google COLAB

---

## Unhandled Exception

When the exception is raised in the program and the program does not handle it, it will be sent to the system for processing which will result in abnormal termination of the program. The program ceases to continue its execution as demonstrated in the following program. In the following example, ‘ZeroDivisionError’ is thrown in statement 3 when attempting to divide a no. by zero which the system will not handle and is ultimately processed by the runtime system.



```
a = 10
b = 0
c = a/b
print(c)

-----
ZeroDivisionError Traceback (most recent call last)
<ipython-input-1-16fdd7ba8379> in <module>()
      1 a = 10
      2 b = 0
----> 3 c = a/b
      4 print(c)

ZeroDivisionError: division by zero
```

SEARCH STACK OVERFLOW

## Exception Handling

The following program is re-written version of above program for exception handling. All the statements to be monitored for exception and enclosed within ‘try’ block which is succeeded by ‘except’ block. If the exception is raised in ‘try’ then it will be handled in the succeeding except block and the program execution continues normally. If the exception is not raised in ‘try’ block then ‘except’ block is skipped and the program execution continues normally. In either case program execution continues normally as demonstrated in the following programs. The structure of exception handling block is shown below:

# Conceptualizing Python in Google COLAB

---

try:

except ExceptionI:

except ExceptinII:

except ExceptionN:

else:

```
try:  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except:  
    print("Cannot Divide with Zero")  
print("Normal Exit")
```

```
Cannot Divide with Zero  
Normal Exit
```

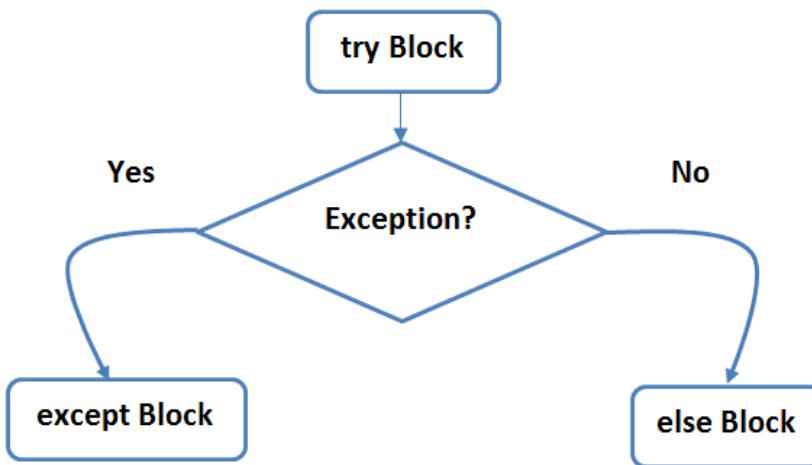
## else Block

The program also can contain '**else**' block which is executed if the exception is not raised in the '**try**' block as shown in the following program:

# Conceptualizing Python in Google COLAB

```
try:  
    a = 10  
    b = 5  
    c = a/b  
    print(c)  
except:  
    print("Cannot Divide with Zero")  
else:  
    print("Division Successful")  
print("Normal Exit")  
  
2.0  
Division Successful  
Normal Exit
```

```
try:  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except:  
    print("Cannot Divide with Zero")  
else:  
    print("Division Successful")  
print("Normal Exit")  
  
Cannot Divide with Zero  
Normal Exit
```



# Conceptualizing Python in Google COLAB

---

Here are few important points about the above-mentioned syntax –

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception (catch all block).
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

## Displaying Descriptive Error Messages

When an exception is raised inn the program, an object of appropriate exception type is instantiated and made available to the program. The program can access the exception object using the following syntax:

*except <exception\_class> as <instance\_name>*

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

The exception instance can be employed for displaying descriptive error messages as shown in the following program:

# Conceptualizing Python in Google COLAB

```
try:  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except Exception as e:  
    print("Cannot Divide with Zero")  
    print(e)  
print("Normal Exit")
```

```
Cannot Divide with Zero  
division by zero  
Normal Exit
```

## Handling Multiple Errors

More than one type of exception can be raised in a ‘try’ block and the program can contain different except blocks for processing exception of each category as demonstrated in the following program:

```
[7] try:  
    fh = open("nofile", "r")  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except Exception as e:  
    print("Generic Exception")  
    print(e)  
except IOError as e:  
    print("IO Error")  
    print(e)  
print("Normal Exit")  
  
Generic Exception  
[Errno 2] No such file or directory: 'nofile'  
Normal Exit
```

# Conceptualizing Python in Google COLAB

```
try:  
    fh = open("nofile", "r")  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except IOError as e:  
    print("IO Error")  
    print(e)  
except Exception as e:  
    print("Generic Error")  
    print(e)  
print("Normal Exit")  
  
IO Error  
[Errno 2] No such file or directory: 'nofile'  
Normal Exit
```

Note: If the generic except block, ‘Exception’ is placed at the beginning, all exceptions get processed there and the succeeding exception blocks become unreachable.

## Default except

When the ‘except’ is used without specifying the name of the Exception class. It is referred to as ‘**default except**’. Default ‘*except:*’ block, if present must be the last except block, otherwise the compiler error is generated as shown in the following program:

```
try:  
    fh = open("nofile", "r")  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except:  
    print("Cannot Divide with Zero")  
except IOError as e:  
    print("IO Error")  
    print(e)  
print("Normal Exit")  
  
File "<ipython-input-9-28f2066a897b>", line 6  
    print(c)  
    ^  
SyntaxError: default 'except:' must be last
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

Python provides another syntax for handling multiple exceptions which is shown below:

Syntax:

```
except(exception_name1, exception_name2, . . . , exception_nameN)
```

All the anticipated exceptions can be enclosed in a pair of parentheses using comma delimiter.

```
try:  
    fh = open("nofile", "r")  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except (IOError, Exception) as e:  
    print(e)  
print("Normal Exit")  
  
[Errno 2] No such file or directory: 'nofile'  
Normal Exit
```

```
try:  
    fh = open("nofile", "r")  
    a = 10  
    b = 0  
    c = a/b  
    print(c)  
except (Exception, IOError ) as e:  
    print(e)  
print("Normal Exit")  
  
[Errno 2] No such file or directory: 'nofile'  
Normal Exit
```

Note: The order of exceptions does not matter.

# Conceptualizing Python in Google COLAB

## finally Demo

‘try’ or ‘except’ blocks can be immediately preceded by ‘finally’ block which is guaranteed to be executed under all circumstances and includes a must execute code. finally block is executed under the following conditions:

- When exception is not thrown
- When exception is thrown
- When function returns

The different cases are depicted below:

```
def test_finally(var):
    try:
        x=int(var)
        print(x)
        return
    except ValueError as Argument:
        print("The argument does not contain numbers\n",Argument)
    finally:
        print("End of Program")

test_finally("100")
test_finally("xyz")
```

```
100
End of Program
The argument does not contain numbers
  invalid literal for int() with base 10: 'xyz'
End of Program
```

# Conceptualizing Python in Google COLAB

## Case 1: No Exception is Raised

```
try:  
    print("Inside try")  
    result=10/5  
except ZeroDivisionError:  
    print("Division By Zero")  
except IOError:  
    print("Cannot Open File")  
except NameError:  
    print("Variale Not Defined")  
except Exception:  
    print("Uknown Error")  
else:  
    print("Inside else")  
finally:  
    print("Inside finally")  
print("Exiting...")
```

```
▶ Inside try  
Inside else  
Inside finally  
Exiting...
```

## Case 2: ‘Division By Zero’ Exception is thrown in ‘try’ Block.

```
try:  
    print("Inside try")  
    result=10/0  
except ZeroDivisionError:  
    print("Division By Zero")  
except IOError:  
    print("Cannot Open File")  
except NameError:  
    print("Variale Not Defined")  
except Exception:  
    print("Uknown Error")  
else:  
    print("Inside else")  
finally:  
    print("Inside finally")  
print("Exiting...")
```

```
▶ Inside try  
Division By Zero  
Inside finally  
Exiting...
```

# Conceptualizing Python in Google COLAB

Case 3: ‘NameError’ Exception is thrown in ‘try’ Block.

```
try:  
    print("Inside try")  
    x=y  
except ZeroDivisionError:  
    print("Division By Zero")  
except IOError:  
    print("Cannot Open File")  
except NameError:  
    print("Variale Not Defined")  
except Exception:  
    print("Unknown Error")  
else:  
    print("Inside else")  
finally:  
    print("Inside finally")  
print("Exiting...")
```

→ Inside try  
Variale Not Defined  
Inside finally  
Exiting...

Case 4: ‘KeyError’ Exception is thrown in ‘try’ Block.

```
try:  
    print("Inside try")  
    dict={"rollno":1,"name":"Maya"}  
    print(dict["division"])  
except ZeroDivisionError:  
    print("Division By Zero")  
except IOError:  
    print("Cannot Open File")  
except NameError:  
    print("Variale Not Defined")  
except Exception:  
    print("Unknown Error")  
else:  
    print("Inside else")  
finally:  
    print("Inside finally")  
print("Exiting...")
```

Inside try  
Uknown Error  
Inside finally  
Exiting...

# Conceptualizing Python in Google COLAB

---

The following table different cases summarized above:

<i>try Block</i>	<i>except ZeroDivisionError Block</i>	<i>except IOError Block</i>	<i>except NameError Block</i>	<i>except Exception Block</i>	<i>else Block</i>	<i>finally Block</i>	<i>Next Statement</i>
No Exception							
result = 10 / 0							
x=y							
KeyError							

## Nested try Blocks

One ‘*try*’ block can be completely nested inside another ‘*try*’ block and this can continue upto any level. In such a case, if the exception is thrown in inner-most try block, all the matching except blocks are examined first, if the matching except block is found, the exception is processed and the program execution continues, otherwise all the ‘except’ blocks corresponding to outer ‘try’ are examined. This continues till outermost ‘except’ blocks are examined. If the matching catch bloc is not found, then ultimately the exception is processed by the runtime system which results in abnormal termination of the program.

Note: If the exception is raised in outer ‘try’ block and matching inner ‘except’ block is present, then the exception is not processed there.

In the following program, ‘*ArithmetricError*’ is raised in the outer ‘*try*’ block which is handled in outer ‘*except*’ block.

# Conceptualizing Python in Google COLAB

```
try:  
    list1=[1]  
    a=10/0  
    it=iter(list1)  
    next(it)  
    next(it)  
    try:  
        a=10/0  
        fh = open("nofile", "r")  
        x=int("xyz")  
        xx=yy  
    except(ArithmetricError, StopIteration, ValueError) as e:  
        print("Caught in Inner except",e)  
    except(ArithmetricError, NameError) as e:  
        print("Caught in Outer except",e)
```

Caught in Outer except division by zero

In the following program, ‘*StopIteration*’ exception is raised in the outer ‘try’ block, but there is no matching outer ‘*except*’ block. Hence is handled by the runtime system.

Note: Even if the inner except block is in place for handling ‘*StopIteration*’ exception, it is not processed there.

```
try:  
    list1=[1]  
    a=10/5  
    it=iter(list1)  
    next(it)  
    next(it)  
    try:  
        a=10/0  
        fh = open("nofile", "r")  
        x=int("xyz")  
        xx=yy  
    except(ArithmetricError, StopIteration, ValueError) as e:  
        print("Caught in Inner except",e)  
    except(ArithmetricError, NameError) as e:  
        print("Caught in Outer except",e)
```

-----  
StopIteration Traceback (most recent call last)  
<ipython-input-16-877d01b298b3> in <module>()  
 4 it=iter(list1)  
 5 next(it)  
----> 6 next(it)  
 7 try:  
 8 a=10/0  
  
StopIteration:

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

In the following program, ‘**ArithmetError**’ is raised in inner ‘**try**’ block which is processed in exception

Both inner and outer except blocks are in place for handling ‘**ArithmetError**’. In such a case the following rule is employed. ‘If the ArithmetError exception is raised in the inner ‘**try**’ block, then it will be handled in inner ‘**except**’ block, and if the ArithmetError exception is raised in the outer ‘**try**’ block, then it will be handled in outer ‘**except**’ block’

```
try:  
    list1=[1]  
    a=10/5  
    it=iter(list1)  
    next(it)  
    #next(it)  
    try:  
        a=10/0  
        fh = open("nofile", "r")  
        x=int("xyz")  
        xx=yy  
    except(ArithmetError, StopIteration, ValueError) as e:  
        print("Caught in Inner except",e)  
    except(ArithmetError, NameError) as e:  
        print("Caught in Outer except",e)  
  
Caught in Inner except division by zero
```

In the following program, ‘FileNotFoundError’ exception is raised in the inner ‘try’ block. There is not inner or outer except block for handling the exception. Hence it will be handled by the runtime system.

# Conceptualizing Python in Google COLAB

```
try:
    list1=[1]
    a=10/5
    it=iter(list1)
    next(it)
    #next(it)
    try:
        a=10/5
        fh = open("nofile", "r")
        x=int("xyz")
        xx=yy
    except(ArithmetricError, StopIteration, ValueError) as e:
        print("Caught in Inner except",e)
    except(ArithmetricError, NameError) as e:
        print("Caught in Outer except",e)

FileNotFoundError                         Traceback (most recent call last)
<ipython-input-19-6aa7113507d6> in <module>()
      7     try:
      8         a=10/5
----> 9         fh = open("nofile", "r")
     10         x=int("xyz")
     11         xx=yy

FileNotFoundError: [Errno 2] No such file or directory: 'nofile'
```

SEARCH STACK OVERFLOW

In the following program ‘*ValueError*’ exception is raised in the inner ‘*try*’ block which is processed in the inner ‘*except*’ block.

```
try:
    list1=[1]
    a=10/5
    it=iter(list1)
    next(it)
    #next(it)
    try:
        a=10/5
        #fh = open("nofile", "r")
        x=int("xyz")
        xx=yy
    except(ArithmetricError, StopIteration, ValueError) as e:
        print("Caught in Inner except",e)
    except(ArithmetricError, NameError) as e:
        print("Caught in Outer except",e)

Caught in Inner except invalid literal for int() with base 10: 'xyz'
```

In the following program, ‘*NameError*’ is raised in the inner ‘*try*’ block and there is no inner ‘*except*’ block for handling it. However, there is outer ‘*except*’ block present for handling ‘*NameError*’. Hence the error is processed in outer ‘*except*’ block.

# Conceptualizing Python in Google COLAB

```
try:  
    list1=[1]  
    a=10/5  
    it=iter(list1)  
    next(it)  
    #next(it)  
    try:  
        a=10/5  
        #fh = open("nofile", "r")  
        #x=int("xyz")  
        xx=yy  
    except(ArithmError, StopIteration, ValueError) as e:  
        print("Caught in Inner except",e)  
    except(ArithmError, NameError) as e:  
        print("Caught in Outer except",e)  
  
Caught in Outer except name 'yy' is not defined
```

## Control Flow in Nested try Blocks

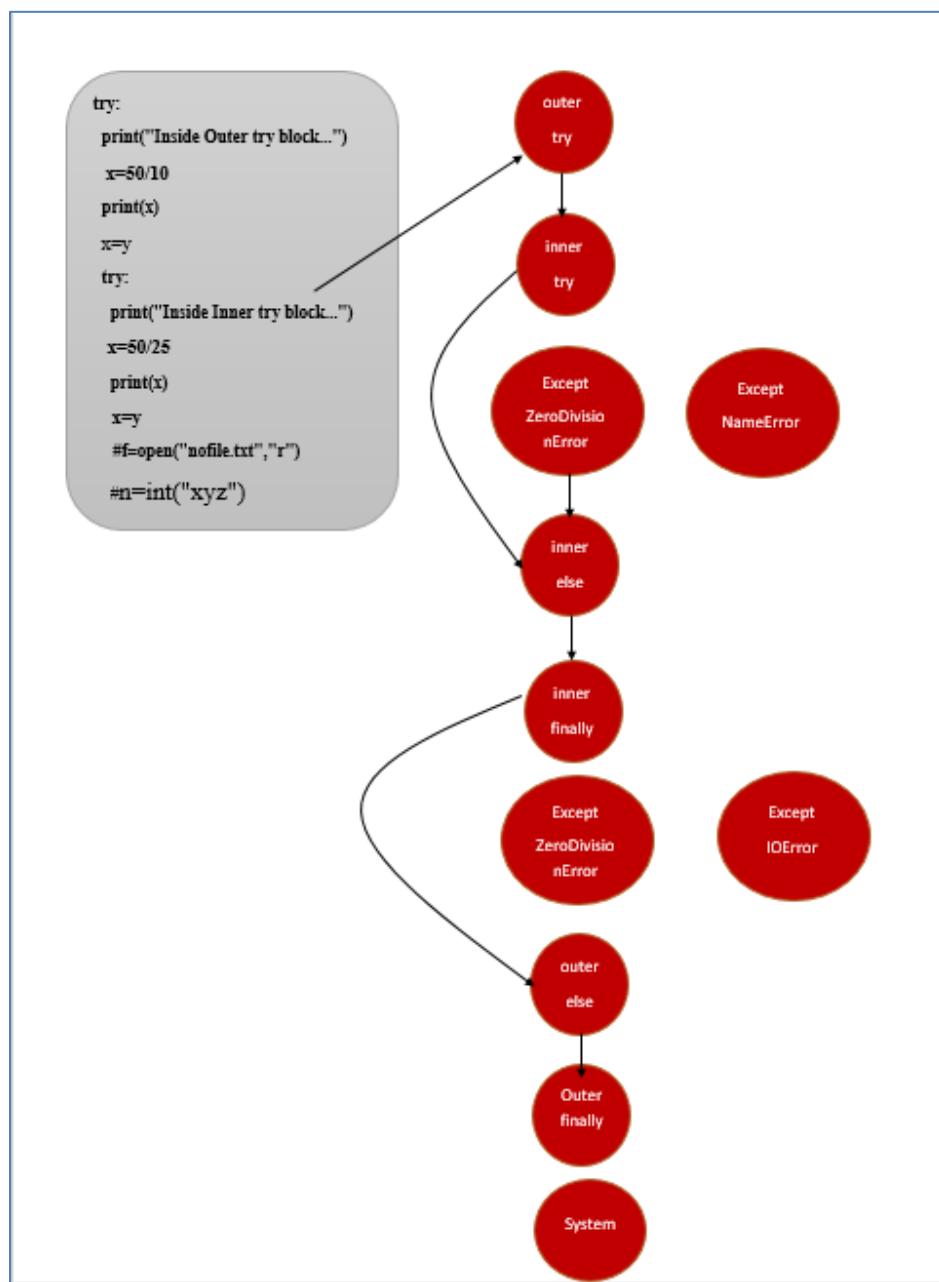
### Case 1: No Exception is raised

```
try:  
    print("Inside Outer try block...")  
    x=50/10  
    print(x)  
    #x=y  
    try:  
        print("Inside Inner try block...")  
        x=50/25  
        print(x)  
        #x=y  
        #f=open("nofile.txt","r")  
        #n=int("xyz")  
    except ZeroDivisionError:  
        print("Zero Division Error Processed in inner except block...")  
    except NameError:  
        print("Name Error Processed in inner except block...")  
    else:  
        print("Inside Inner else block...")  
    finally:  
        print("Inside Inner finally block...")  
except ZeroDivisionError:  
    print("Zero Division Error Processed in outer except block...")  
except IOError:  
    print("I/O Error Processed in outer except block...")  
else:  
    print("Inside Outer else block...")  
finally:  
    print("Inside Outer finally block...")
```

# Conceptualizing Python in Google COLAB

On execution the following output is generated:

```
Inside Outer try block...
5.0
Inside Inner try block...
2.0
Inside Inner else block...
Inside Inner finally block...
Inside Outer else block...
Inside Outer finally block...
```



# Conceptualizing Python in Google COLAB

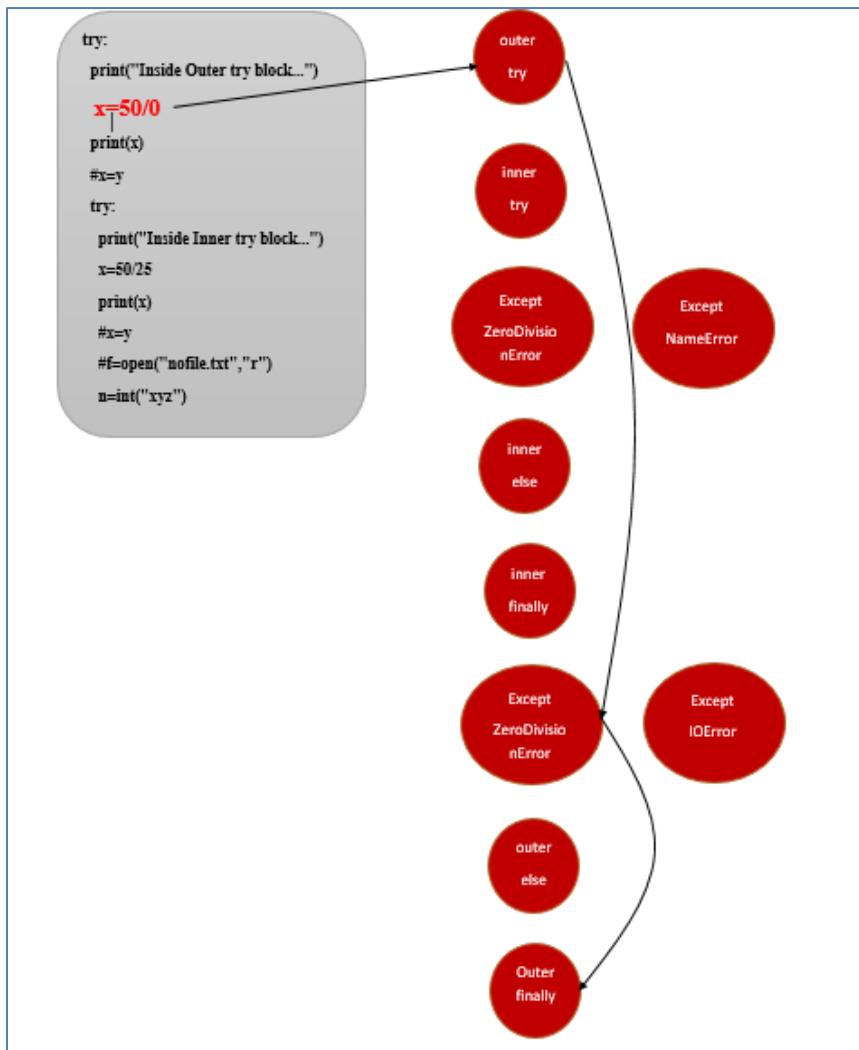
## Case 2: ZeroDivisionError Exception is raised in Outer try Block

Modify outer try block as shown below:

```
try:  
    print("Inside Outer try block...")  
    x=50/0  
    print(x)  
    #x=y
```

On execution the following output is generated:

```
Inside Outer try block...  
Zero Division Error Processed in outer except block...  
Inside Outer finally block...
```



# Conceptualizing Python in Google COLAB

## Case 3: ZeroDivisionError Exception is raised in Inner try Block

Modify outer and inner try blocks as shown below:

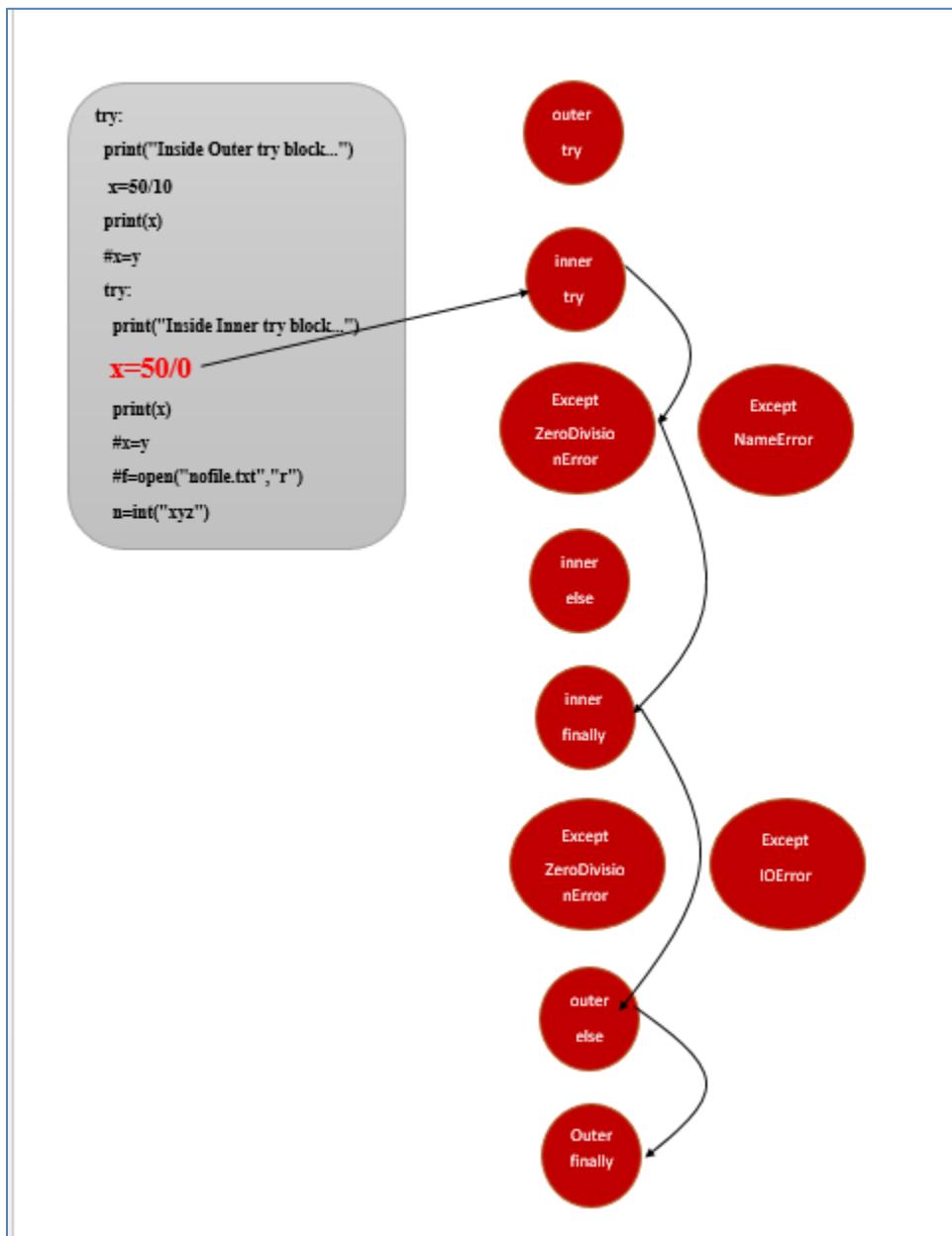


```
try:  
    print("Inside Outer try block...")  
    x=50/10  
    print(x)  
    #x=y  
    try:  
        print("Inside Inner try block...")  
        x=50/0  
        print(x)  
        #x=y  
        #f=open("nofile.txt", "r")  
        #n=int("xyz")
```

On execution the following output is generated:

```
Inside Outer try block...  
5.0  
Inside Inner try block...  
Zero Division Error Processed in inner except block...  
Inside Inner finally block...  
Inside Outer else block...  
Inside Outer finally block...
```

# Conceptualizing Python in Google COLAB



Case 4: `NameError` Exception is raised in Outer try Block

Modify outer and inner try blocks as shown below:

# Conceptualizing Python in Google COLAB

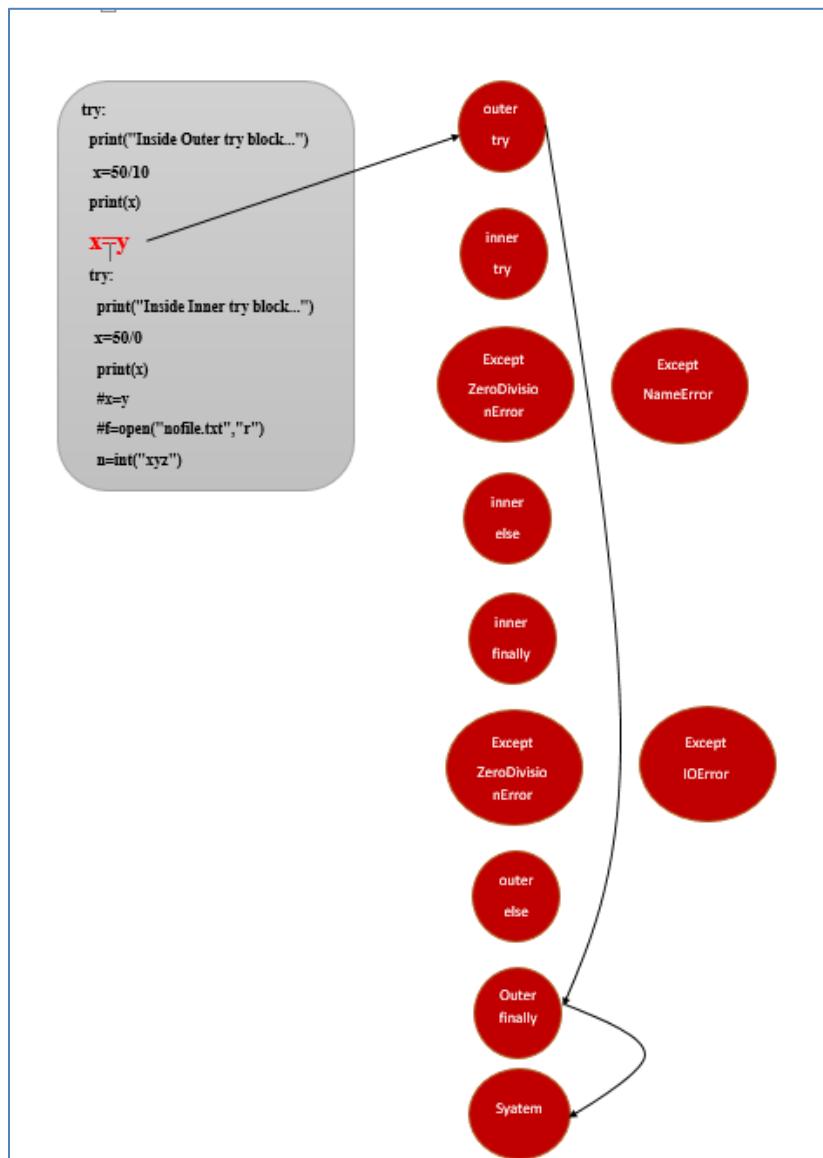
```
try:  
    print("Inside Outer try block...")  
    x=50/10  
    print(x)  
    x=y  
    try:  
        print("Inside Inner try block...")  
        x=50/25  
        print(x)  
        #x=y  
        #f=open("nofile.txt","r")  
        #n=int("xyz")  
    except ZeroDivisionError:  
        print("Zero Division Error Processed in inner except block...")  
    except NameError:  
        print("Name Error Processed in inner except block...")  
    else:  
        print("Inside Inner else block...")  
    finally:  
        print("Inside Inner finally block...")  
except ZeroDivisionError:  
    print("Zero Division Error Processed in outer except block...")  
except IOError:  
    print("I/O Error Processed in outer except block...")  
else:  
    print("Inside Outer else block...")  
finally:  
    print("Inside Outer finally block...")
```

On execution the following output is generated:

```
Inside Outer try block...  
5.0  
Inside Outer finally block...  
-----  
NameError Traceback (most recent call last)  
<ipython-input-13-d80d1b7c188d> in <module>()  
      3     x=50/10  
      4     print(x)  
----> 5     x=y  
      6     try:  
      7         print("Inside Inner try block...")  
  
NameError: name 'y' is not defined
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB



## Case 5: NameError Exception is raised in Inner try Block

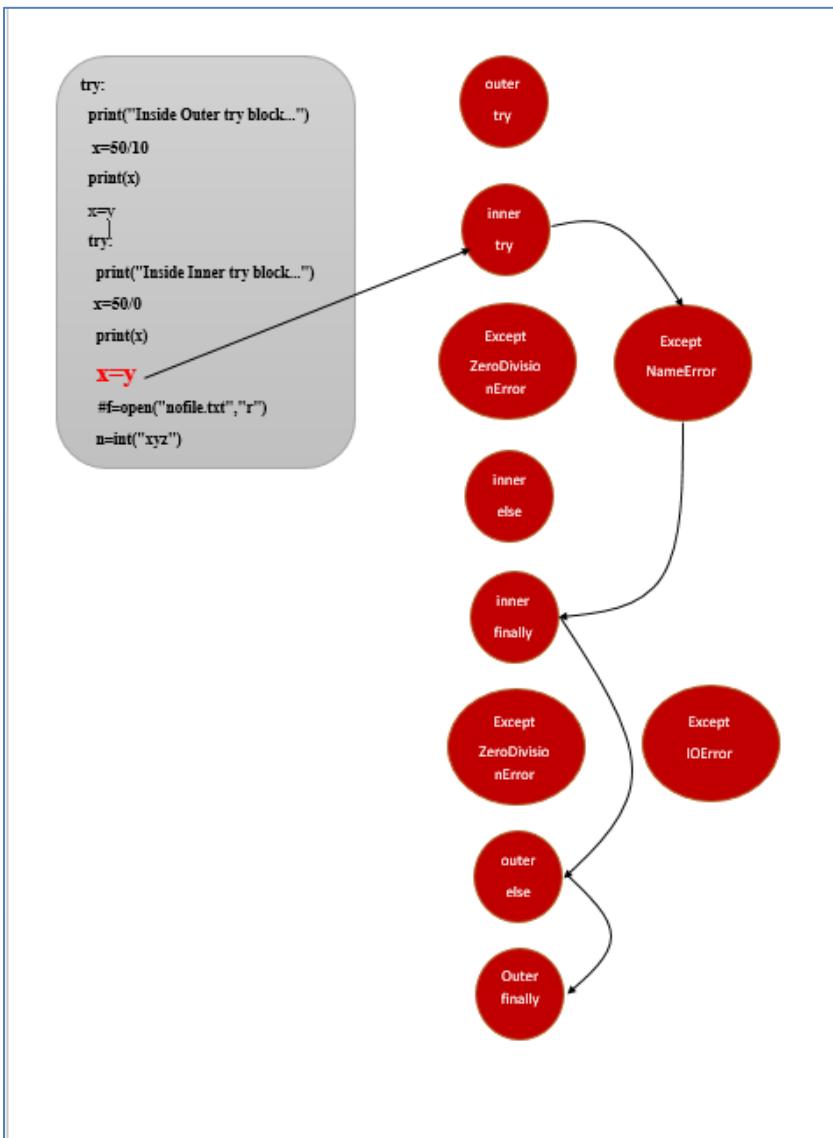
Modify outer and inner try blocks as shown below:

```
try:  
    print("Inside Outer try block...")  
    x=50/10  
    print(x)  
    #x=y  
    try:  
        print("Inside Inner try block...")  
        x=50/25  
        print(x)  
        x=y  
        #f=open("nofile.txt","r")  
        n=int("xyz")
```

# Conceptualizing Python in Google COLAB

On execution the following output is generated:

```
Inside Outer try block...
5.0
Inside Inner try block...
2.0
Name Error Processed in inner except block...
Inside Inner finally block...
Inside Outer else block...
Inside Outer finally block...
```



# Conceptualizing Python in Google COLAB

---

## Case 6: IOError Exception Raised in Inner try Block

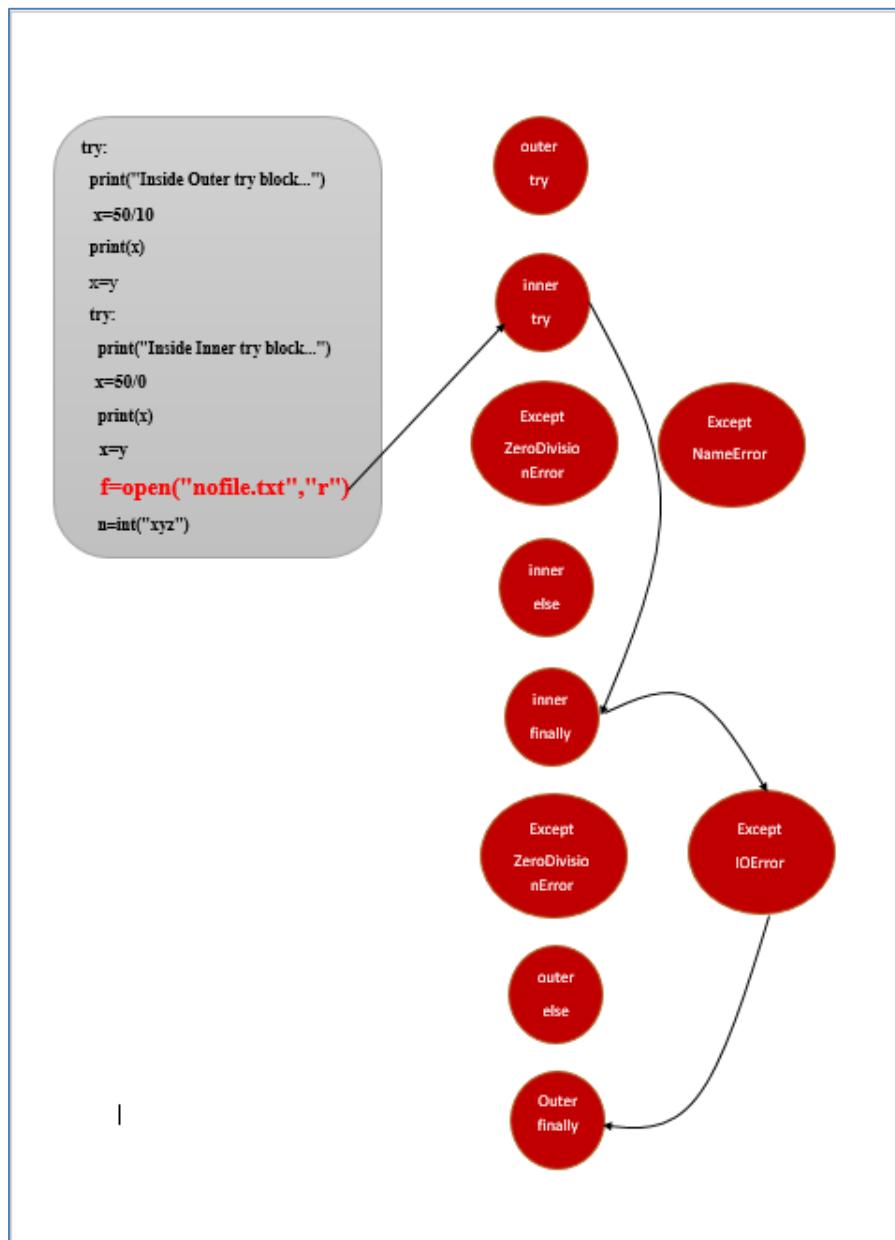
Modify outer and inner try blocks as shown below:

```
try:  
    print("Inside Outer try block...")  
    x=50/10  
    print(x)  
    #x=y  
    try:  
        print("Inside Inner try block...")  
        x=50/25  
        print(x)  
        #x=y  
        f=open("nofile.txt","r")  
        #n=int("xyz")
```

On execution the following output is generated:

```
Inside Outer try block...  
5.0  
Inside Inner try block...  
2.0  
Inside Inner finally block...  
I/O Error Processed in outer except block...  
Inside Outer finally block...
```

# Conceptualizing Python in Google COLAB



## Case 7: ValueError Exception Raised in Inner try Block

Modify outer and inner try blocks as shown below:

# Conceptualizing Python in Google COLAB

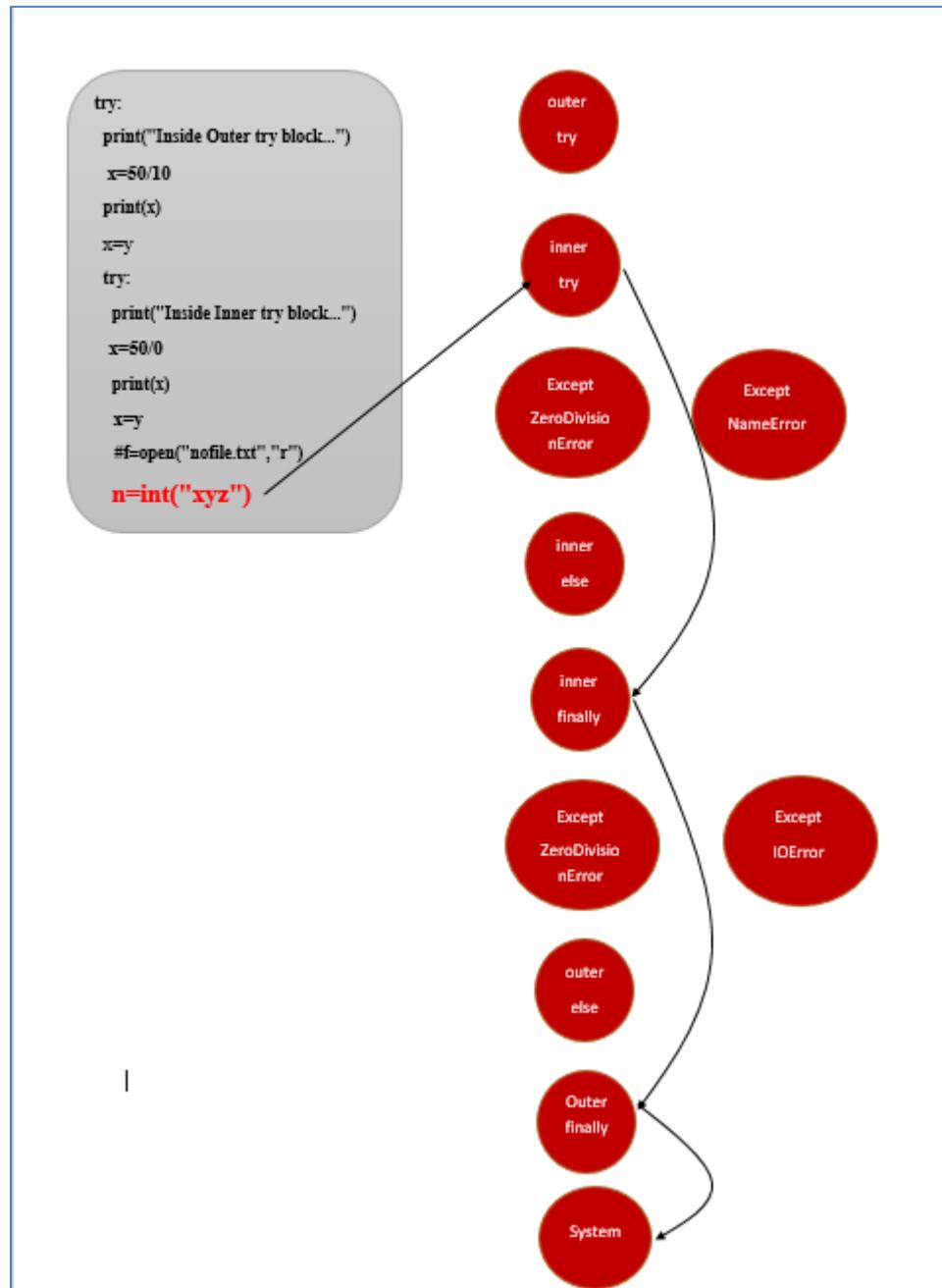
```
try:  
    print("Inside Outer try block...")  
    x=50/10  
    print(x)  
    #x=y  
    try:  
        print("Inside Inner try block...")  
        x=50/25  
        print(x)  
        #x=y  
        #f=open("nofile.txt","r")  
        n=int("xyz")
```

On execution the following output is generated:

```
Inside Outer try block...  
5.0  
Inside Inner try block...  
2.0  
Inside Inner finally block...  
Inside Outer finally block...  
-----  
ValueError Traceback (most recent call last)  
<ipython-input-16-59d909f4df81> in <module>()  
      10     #x=y  
      11     #f=open("nofile.txt","r")  
----> 12     n=int("xyz")  
      13     except ZeroDivisionError:  
      14         print("Zero Division Error Processed in inner except block...")  
  
ValueError: invalid literal for int() with base 10: 'xyz'
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB



# Conceptualizing Python in Google COLAB

## Valid Exception Handling Structures

Rule 1: ‘try’ block must be followed by either ‘*except*’ or ‘*finally*’ block.

A screenshot of a Google Colab cell. The code is:

```
try:  
    print("10")
```

The line `print("10")` is underlined in red. Below the code, the output shows:

```
File "<ipython-input-22-c7c1dbe7293b>", line 2  
    print("10")  
          ^  
SyntaxError: unexpected EOF while parsing
```

At the bottom of the cell is a button labeled "SEARCH STACK OVERFLOW".

A screenshot of a Google Colab cell. The code is:

```
try:  
    print(10)  
finally:  
    print("End")
```

The output shows:

```
10  
End
```

Rule 2: ‘else’ block if present should be placed before ‘finally’ block.

A screenshot of a Google Colab cell. The code is:

```
try:  
    print(10)  
except:  
    print("Error")  
finally:  
    print("End")  
else:  
    print("No Error")
```

The line `else:` is underlined in red. Below the code, the output shows:

```
File "<ipython-input-26-052a494e89f0>", line 7  
    else:  
        ^  
SyntaxError: invalid syntax
```

At the bottom of the cell is a button labeled "SEARCH STACK OVERFLOW".

# Conceptualizing Python in Google COLAB

Rule 3: The default except block, if present, should be the last 'except' block.

Find Output:

The screenshot shows a code cell in Google Colab. The code is:

```
try:  
    x=int("xxx")  
except:  
    x=10/0  
except ZeroDivisionError:  
    print("Zero Divsion Error")
```

The output shows a SyntaxError:

```
File "<ipython-input-27-8804b0c59968>", line 2  
    x=int("xxx")  
    ^  
SyntaxError: default 'except:' must be last
```

A button labeled "SEARCH STACK OVERFLOW" is visible at the bottom of the code cell.

The screenshot shows a code cell in Google Colab. The code is:

```
try:  
    x=int("xxx")  
except ValueError:  
    print("Caught: Value Error")  
    x=10/0  
except:  
    print("Division By Zero Error")
```

The output shows a stack trace for a ValueError:

```
Caught: Value Error  
-----  
ValueError Traceback (most recent call last)  
<ipython-input-28-4e9a767a760e> in <module>()  
      1 try:  
----> 2     x=int("xxx")  
      3 except ValueError:  
  
ValueError: invalid literal for int() with base 10: 'xxx'  
  
During handling of the above exception, another exception occurred:  
  
ZeroDivisionError Traceback (most recent call last)  
<ipython-input-28-4e9a767a760e> in <module>()  
      3 except ValueError:  
----> 4     print("Caught: Value Error")  
      5     x=10/0  
      6 except:  
      7     print("Division By Zero Error")  
  
ZeroDivisionError: division by zero
```

# Conceptualizing Python in Google COLAB

---

**Rule 4:** When more than one exception is specified in a single ‘except’ block, all exceptions must be enclosed within a pair of parentheses.

The screenshot shows a Google Colab cell with the following code:

```
try:  
    a=10/0  
except ArithmeticError, IOError:  
    print("Arithmetc Exception")  
else:  
    print("Successfully Done")
```

An error message is displayed below the code:

```
File "<ipython-input-29-30a0c08e36ac>", line 3  
    except ArithmeticError, IOError:  
                           ^  
SyntaxError: invalid syntax
```

A button labeled "SEARCH STACK OVERFLOW" is visible at the bottom of the cell.

Note: When multiple exceptions are specified, brackets are required.

The screenshot shows a Google Colab cell with the following code:

```
try:  
    a=10/0  
except (ArithmeticError, IOError):  
    print("Arithmetc Exception")  
else:  
    print("Successfully Done")
```

The output of the code is:

```
Arithmetc Exception
```

What will be the output of the following Python code?

```
def foo():  
    try:  
        return 1  
    finally:  
        return 2  
k = foo()
```

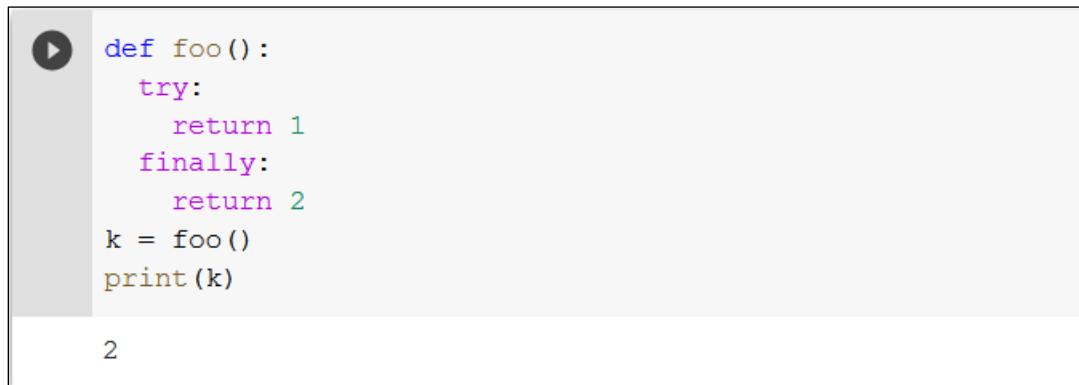
# Conceptualizing Python in Google COLAB

---

*print(k)*

The finally block is executed even there is a return statement in the try block.

On execution of the above program, the following output is generated:



```
def foo():
    try:
        return 1
    finally:
        return 2
k = foo()
print(k)
```

2

What will be the output of the following Python code?

*def foo():*

*try:*

*print(1)*

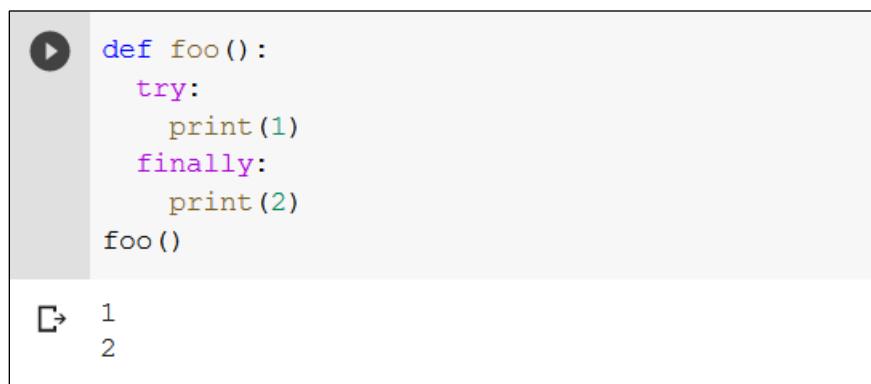
*finally:*

*print(2)*

*foo()*

No error occurs in the try block so 1 is printed. Then the finally block is executed and 2 is printed.

On execution of the above program, the following output is generated:



```
def foo():
    try:
        print(1)
    finally:
        print(2)
foo()
```

1  
2

# Conceptualizing Python in Google COLAB

---

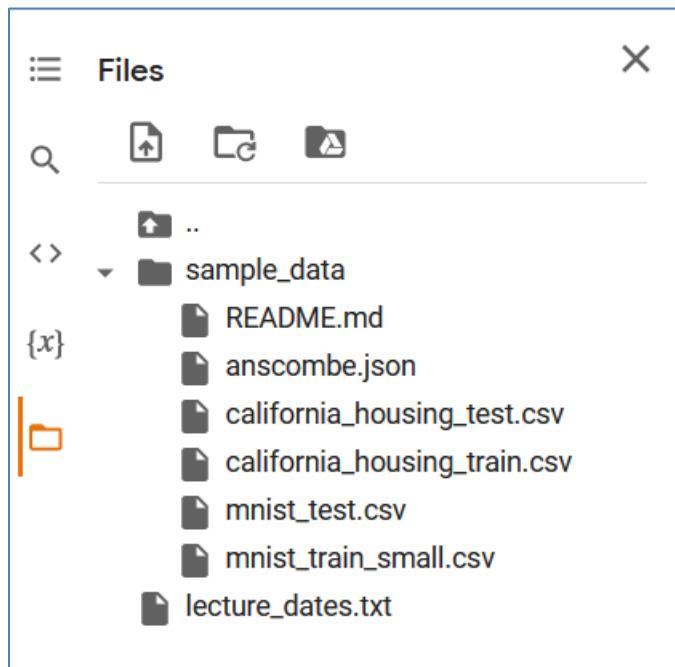
## Chapter 9

### Lab Assignment on File Handling in Python

Level – Intermediate

#### Uploading the File in Colab

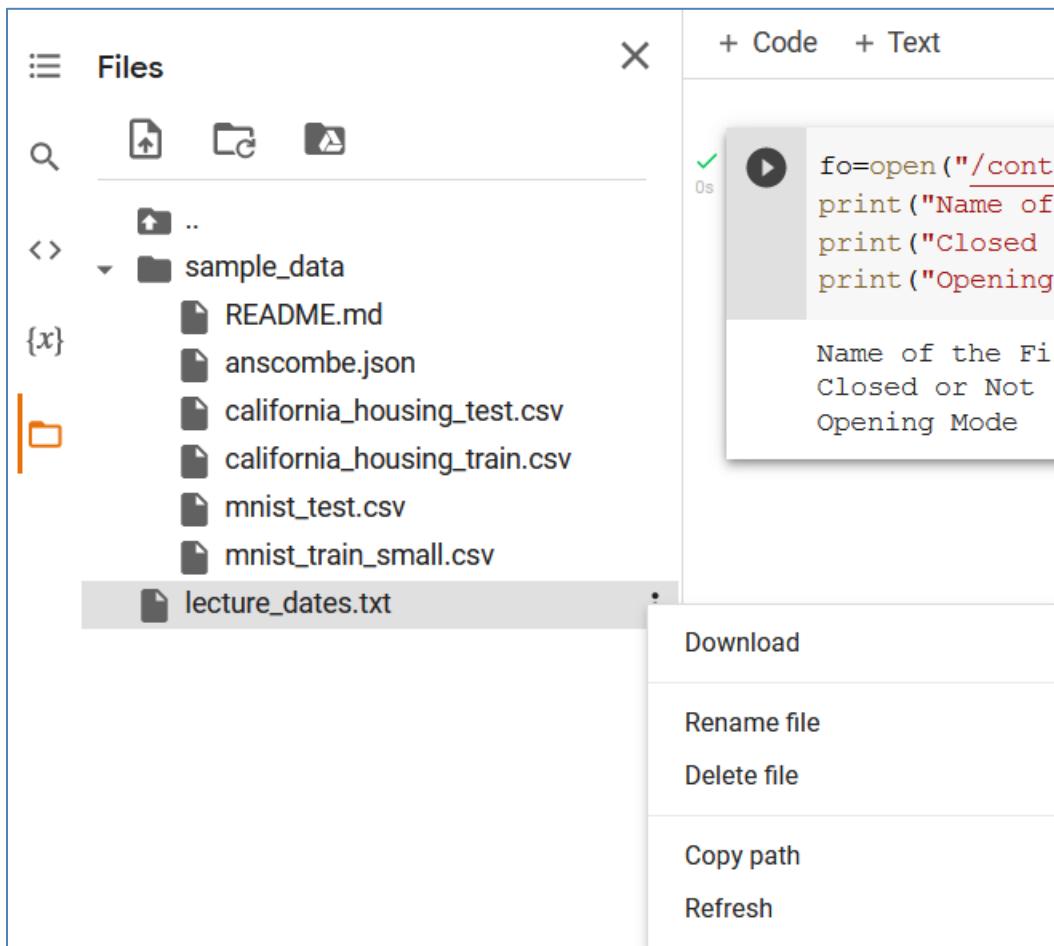
For uploading the file in Colab, click on the ‘**Upload File**’ icon, browse to the local file system and select the file to be uploaded as shown in the following figure:



#### Copying File Path

For coping the path of the file, select and right-click on the file in Colab and select ‘**Copy path**’ option from the shortcut menu as shown below:

# Conceptualizing Python in Google COLAB



## Basic File Operations

### Opening the File for IO

Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a file object, which would be utilized to call other support methods associated with it. The syntax of *open()* function is shown below:

```
file_object = open(file_name,[,access_mode],[,buffering])
```

The meaning of different parameters is depicted in the following table:

# Conceptualizing Python in Google COLAB

Parameter	Meaning
file_name	A string value containing the name of the file to be accessed.
access_mode	It determines the mode in which the file needs to be opened which depends on the type of operation to be performed on a file. This is optional parameter and the default file access mode is read (r).
buffering	It can take positive, negative or zero values. The default is negative which refers to the system default. If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size.

## File Access Modes

Here is a list of the different modes of opening a file –

Sr.No.	Modes & Description
1	<b>r</b> Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	<b>rb</b> Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	<b>r+</b> Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	<b>rb+</b> Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.

# Conceptualizing Python in Google COLAB

---

5	<b>w</b> Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	<b>wb</b> Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	<b>w+</b> Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	<b>wb+</b> Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	<b>a</b> Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	<b>ab</b> Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	<b>a+</b> Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	<b>ab+</b> Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

# Conceptualizing Python in Google COLAB

---

## The *file* Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object –

Sr.No.	Attribute & Description
1	<b>file.closed</b> Returns true if file is closed, false otherwise.
2	<b>file.mode</b> Returns access mode with which file was opened.
3	<b>file.name</b> Returns name of the file.
4	<b>file.softspace</b> Returns false if space explicitly required with print, true otherwise.

The following program demonstrates different file attributes of a file opened using open() function.

```
fo=open("/content/lecture_dates.txt","wb")
print("Name of the File : ",fo.name)
print("Closed or Not    : ",fo.closed)
print("Opening Mode     : ",fo.mode)

Name of the File : /content/lecture_dates.txt
Closed or Not    : False
Opening Mode     : wb
```

# Conceptualizing Python in Google COLAB

---

## Closing the File

### The *close()* Method

The *close()* method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the *close()* method to close a file. The syntax for closing the file is shown below:

### Syntax

*<file\_object>.close()*

Python supports a *tell()* function for querying the position of file pointer. The following program displays the position of the file pointer when the file is opened in different modes specified above for writing to the file and reading the file contents.

```
f=open("test.txt","w")
print(f.tell())
f=open("test.txt","wb")
print(f.tell())
f=open("test.txt","w+")
print(f.tell())
f=open("test.txt","wb+")
print(f.tell())
```

```
0
0
0
0
```

# Conceptualizing Python in Google COLAB



```
f=open("test.txt","r")
print(f.tell())
f=open("test.txt","rb")
print(f.tell())
f=open("test.txt","r+")
print(f.tell())
f=open("test.txt","rb+")
print(f.tell())
```

```
0
0
0
0
```

## FileNotFoundException Exception

*FileNotFoundException* exception is raised when the file is opened in ‘*r*’ mode and the file does not exist at the specified location. If the file is opened in ‘*w*’ mode and the file does not exist, then the new file is created.



```
f=open("noname.txt","r")
f.close()

-----
FileNotFoundException                                Traceback (most recent call last)
<ipython-input-32-f6e7c556f546> in <module>()
----> 1 f=open("noname.txt","r")
      2 f.close()

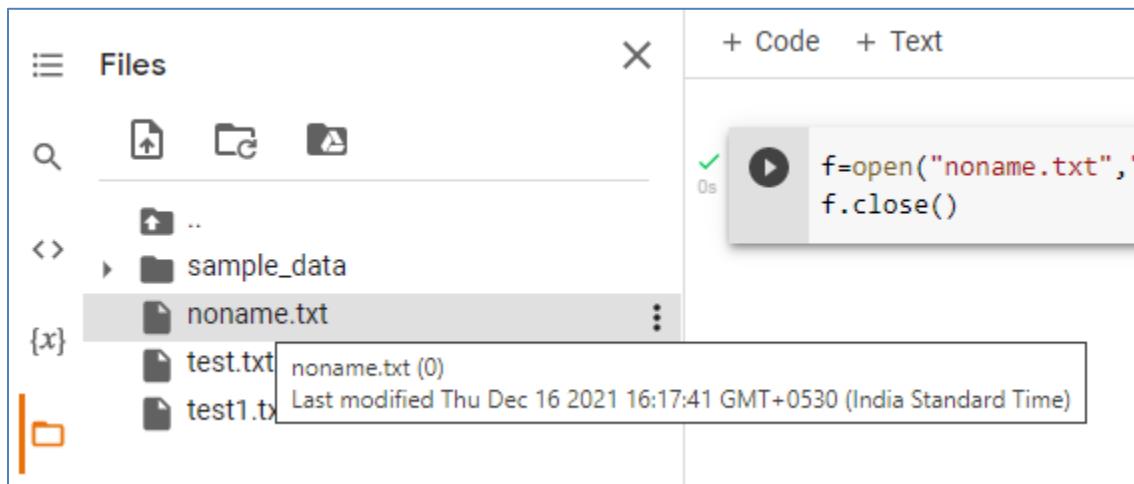
FileNotFoundException: [Errno 2] No such file or directory: 'noname.txt'
```

SEARCH STACK OVERFLOW



```
f=open("noname.txt","w")
f.close()
```

# Conceptualizing Python in Google COLAB



## Creating a File and Writing Content

### File I/O

The file object provides a set of access methods for performing file I/O, for writing contents to the file and reading content from the file.

#### The `write()` Method

The `write()` method writes a string to the file opened in ‘w’, ‘w+’, ‘a’ or ‘a+’ modes which is passed as an argument to it. It is important to note that Python strings can have binary data and not just text.

The `write()` method does not add a newline character ('\n') to the end of the string. The syntax of `write()` method is shown below:

#### Syntax:

```
<file_object>.write(<string>)
```

The following program creates a new file with the name ‘foo.txt’, if the file with that name does not already exist or erases the contents if the file exists and writes the content passed to the `write()` method. The file is closed after the write operation is complete.

# Conceptualizing Python in Google COLAB

```
fo=open("/content/foo.txt","w")
fo.write("Python is great language.\nPython is interesting.\nPython is easy")
print("File created successfully...")
fo.close()
```

```
File created successfully...
```

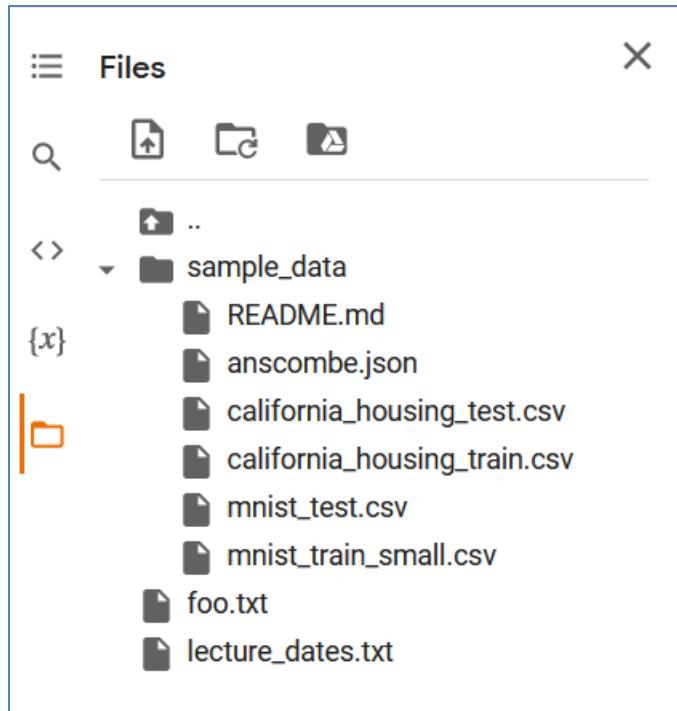
In the following program, the file ‘test.txt’ is opened in ‘w’ mode and the string ‘MCA’ is written to the file.

```
f=open("test.txt","w")
f.write("MCA")
print("MCA written to the file test.txt...")
f.close()
```

```
MCA written to the file test.txt...
```

## Refresh Files

To view the changes in the ‘*Files*’ tab of Colab, click on ‘**Refresh**’ button at the top.



# Conceptualizing Python in Google COLAB

---

## Reading Content of the File

### The `read()` Method

The `read()` method reads a string from an open file. It is important to note that Python strings can have binary data apart from text data. The syntax of `read()` method is shown below:

Syntax:

```
<file_object>.read([<bytes>])
```

Here, the parameter passed to read method specifies the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, until the end of file.

In the following program, the file '`test.txt`' is opened in '`r`' mode and the entire contents of the file are read in a string variable, '`data`'.

A screenshot of a Google Colab interface. On the left, there is a play button icon. The code cell contains the following Python code:

```
f=open("test.txt","r")
data=f.read()
print("test.txt file contains...",data)
f.close()
```

The output cell shows the result of the print statement:

```
test.txt file contains... MCA
```

A screenshot of a Google Colab interface. On the left, there is a play button icon. The code cell contains the following Python code:

```
fo=open("/content/foo.txt", "r+")
str=fo.read(7)
print(str)
fo.close()
```

The output cell shows the result of the print statement:

```
Python
```

## Reading the Entire Content of the File

If no parameter is passed to `read()` method, the entire content of the file is read as demonstrated in the following program:

# Conceptualizing Python in Google COLAB

```
▶ fo=open("/content/foo.txt","r+")
  str=fo.read()
  print(str)
  fo.close()
```

```
Python is great language.
Python is interesting.
Python is easy
```

## Tracking File Pointer

### File Positions

The ***tell()*** method tells you the current position of the file pointer within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The ***seek(offset[, from])*** method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

If from is set to ***0***, it means use the beginning of the file as the reference position and ***1*** means use the current position as the reference position and if it is set to ***2*** then the end of the file would be taken as the reference position.

## Querying Position of File Pointer

The ***tell()*** method of the file object returns the current position of the file pointer from the beginning of the file which specifies the byte from which the next read or next write operation would commence.

```
▶ fo=open("/content/foo.txt","r+")
  position=fo.tell()
  print("Position of File Pointer",position)
  fo.close()
```

```
Position of File Pointer 0
```

# Conceptualizing Python in Google COLAB

## Relocating File Pointer

In the following program seek() method is used for positioning the file pointer at the location 10 bytes from the beginning of the file and then the next 15 bytes from the file are read.

```
▶ fo=open("/content/foo.txt","r+")
  fo.seek(10,0)
  str=fo.read(15)
  print(str)
  fo.close()

great language.
```

As shown in the following program, when the file is opened in ‘r’ mode negative values cannot be specified for end-relative seeks. While it works fine when the file is opened in ‘rb’ mode as demonstrated in the following programs:

```
▶ import os

file1=open("test1.txt","r")
file1.seek(-1,2)
data=file1.read()
print(data)
file1.close()

-----
UnsupportedOperation                                Traceback (most recent call last)
<ipython-input-38-be91bc75848c> in <module>()
      2
      3 file1=open("test1.txt","r")
----> 4 file1.seek(-1,2)
      5 data=file1.read()
      6 print(data)

UnsupportedOperation: can't do nonzero end-relative seeks
```

[SEARCH STACK OVERFLOW](#)

# Conceptualizing Python in Google COLAB

```
▶ file1=open("test1.txt","rb")
  file1.seek(-3,2)
  data=file1.read().decode()
  print(data)
  file1.close()
```

↳ hon

In the following program, the file ‘test.txt’ is opened in ‘w+’ mode for performing read and write operations. After writing ‘MCA’ to the file, the file pointer is positioned at the end of the file. The seek() method is used for relocating the pointer at the beginning of the file and the entire content of the file is read in a variable ‘data’ and the same is printed.

Next, file pointer is moved by one byte and the character ‘C’ in ‘MCA’ is replaced with the character ‘B’.

The file is closed and re-opened in ‘r’ mode and the the entire content of the file is read in a variable ‘data’ and the same is printed.

```
▶ f=open("test.txt","w+")
  f.write("MCA")
  f.seek(0)
  data=f.read()
  print("test.txt file contains...",data)

  f.seek(1)
  f.write("B")
  f.close()

  f=open("test.txt","r")
  data=f.read()
  print("test.txt file now contains...",data)
  f.close()
```

test.txt file contains... MCA  
test.txt file now contains... MBA

# Conceptualizing Python in Google COLAB

```
f=open("test.txt","w")
f.write("MCA")
f.close()

f=open("test.txt","r")
data=f.read()
print("test.txt file contains...",data)
f.close()

f=open("test.txt","r+")
f.seek(1,0)
f.write("B")
f.close()

f=open("test.txt","r")
data=f.read()
print("test.txt file now contains...",data)
f.close()
```

```
test.txt file contains... MCA
test.txt file now contains... MBA
```

## Difference Between w and wb

When the file is opened in ‘**wb**’ mode for writing in binary format. Passing a string to **write()** method of file object throws ‘**TypeError**’ exception as shown in the following figure:

```
import os
file1=open("test1.txt","w")
file1.write("Python")
file1.close()

file2=open("test2.txt","wb")
file2.write("Python")
file2.close()

print(os.path.getsize("test1.txt"))
print(os.path.getsize("test2.txt"))

-----  

TypeError                                 Traceback (most recent call last)
<ipython-input-27-dacf44996180> in <module>()
      5
      6 file2=open("test2.txt","wb")
----> 7 file2.write("Python")
      8 file2.close()
      9

TypeError: a bytes-like object is required, not 'str'
```

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

---

To fix the bug use *encode()* method of ‘str’ class as shown in the following program:

```
▶ import os  
file1=open("test1.txt","w")  
file1.write("Python")  
file1.close()  
  
file2=open("test2.txt","wb")  
file2.write("Python".encode())  
file2.close()  
  
print(os.path.getsize("test1.txt"))  
print(os.path.getsize("test2.txt"))
```

```
⇨ 6  
6
```

## Difference Between r and rb

```
▶ import os  
  
file1=open("test1.txt","r")  
data=file1.read()  
print(data)  
file1.close()  
  
file2=open("test2.txt","rb")  
data=file2.read()  
print(data)  
file2.close()  
  
print(os.path.getsize("test1.txt"))  
print(os.path.getsize("test2.txt"))
```

```
⇨ Python  
b'Python'  
6  
6
```

On reading a binary file using *read()* method, the text written to the file is enclosed within b''. To read the text, use *decode()* method as shown in the following program:

# Conceptualizing Python in Google COLAB

```
import os

file1=open("test1.txt","r")
data=file1.read()
print(data)
file1.close()

file2=open("test2.txt","rb")
data=file2.read().decode()
print(data)
file2.close()

print(os.path.getsize("test1.txt"))
print(os.path.getsize("test2.txt"))
```

```
Python
Python
6
6
```

## Difference Between w and w+

‘w’ mode creates a new file, if the specified file does not exist or truncates the file, if the file already exists.

## Difference Between r and r+

‘r’ mode opens the file in read mode and places the file pointer at the beginning of the file. ‘r+’ mode opens the file for both reading and writing and does not truncate the contents of the file, if the file already exists.

## Difference Between r+ and w+

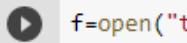
Both ‘r+’ and ‘w+’ modes open the file in read and write mode. The difference is that if the file already exists, ‘w+’ deletes the contents while ‘r+’ does not delete the contents but overwrites it.

In the following program, the following tasks are performed:

- the file ‘test.txt’ is opened in ‘w’ mode for writing the content ‘MCA in SIBER’.
- the file is then closed and re-opened in ‘r’ mode for reading the contents.
- Next, the file is opened in ‘w+’ mode, new content ‘MBA’ is written to the file,

# Conceptualizing Python in Google COLAB

- the file pointer is positioned at the beginning of the file using seek() method and the contents are read and displayed.



```
f=open("test.txt","w")
f.write("MCA in SIBER")
f.close()

f=open("test.txt","r")
data=f.read()
print("test.txt file contains...",data)
f.close()

f=open("test.txt","w+")
f.write("MBA")
f.seek(0)
data=f.read()
print("test.txt file contains...",data)
f.close()
```



test.txt file contains... MCA in SIBER  
test.txt file contains... MBA



```
f=open("test.txt","w")
f.write("MCA in SIBER")
f.close()

f=open("test.txt","r")
data=f.read()
print("test.txt file contains...",data)
f.close()

f=open("test.txt","r+")
f.write("MBA")
f.seek(0)
data=f.read()
print("test.txt file contains...",data)
f.close()
```



test.txt file contains... MCA in SIBER  
test.txt file contains... MBA in SIBER

# Conceptualizing Python in Google COLAB

---

## Append Mode

The append mode enables appending content to the existing content of the file leaving the original content of the file intact. In the following program,

- the string ‘MCA in SIBER’ is written to the file ‘test.txt’ by opening it in ‘w’ mode.
- the file is closed and re-opened in ‘a+’ mode for appending the content ‘MBA in SIBER’.
- the file pointer is re-positioned at the beginning of the file using seek() method and the entire contents of the file are read into a variable ‘data’ and the same is printed.

```
f=open("test.txt","w")
f.write("MCA in SIBER")
f.close()

f=open("test.txt","a+")
f.write(" MBA in SIBER")
f.seek(0)
data=f.read()
print("test.txt file contains...",data)
f.close()
```

```
▶ test.txt file contains... MCA in SIBER MBA in SIBER
```

In text files (those opened without a b in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with seek(0, 2)).

This is because text files do not have a 1-to-1 correspondence between encoded bytes and the characters they represent, so seek can't tell where to jump to in the file to move by a certain number of characters.

If your program is okay with working in terms of raw bytes, you can change your program to read:

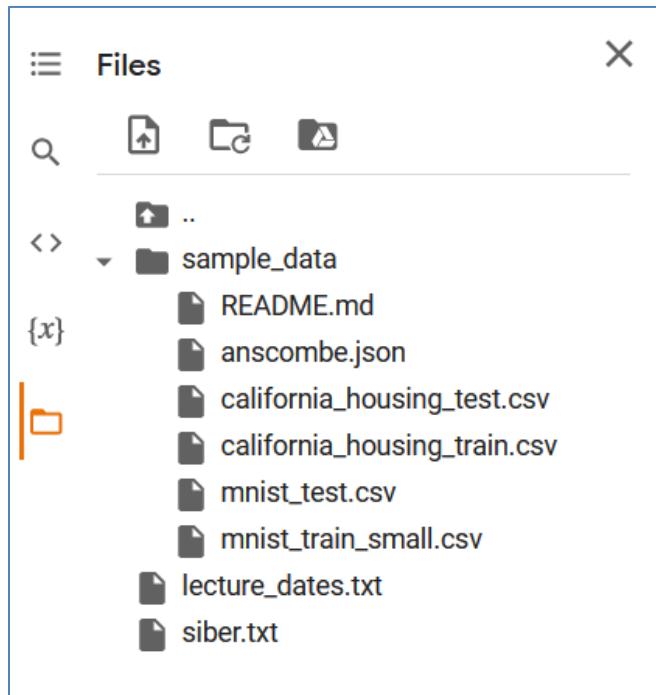
# Conceptualizing Python in Google COLAB

---

```
fo=open("/content/foo.txt","rb")
fo.seek(-7,2)
str=fo.read()
print(str)
fo.close()

b'is easy'
```

Refresh Files



## Managing File System

Python 'os' module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

# Conceptualizing Python in Google COLAB

---

## Renaming a File

### The rename() Method

The rename() method takes two arguments, the current filename and the new filename. The syntax of rename() function is shown below:

#### Syntax

*os.rename(<old\_filename>,<new-filename>)*

```
import os  
os.rename("foo.txt","siber.txt")  
print("File Renamed Successfully...")  
  
File Renamed Successfully...
```

## Deleting a File

### The remove() Method

You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument. The syntax of remove() method is shown below:

#### Syntax

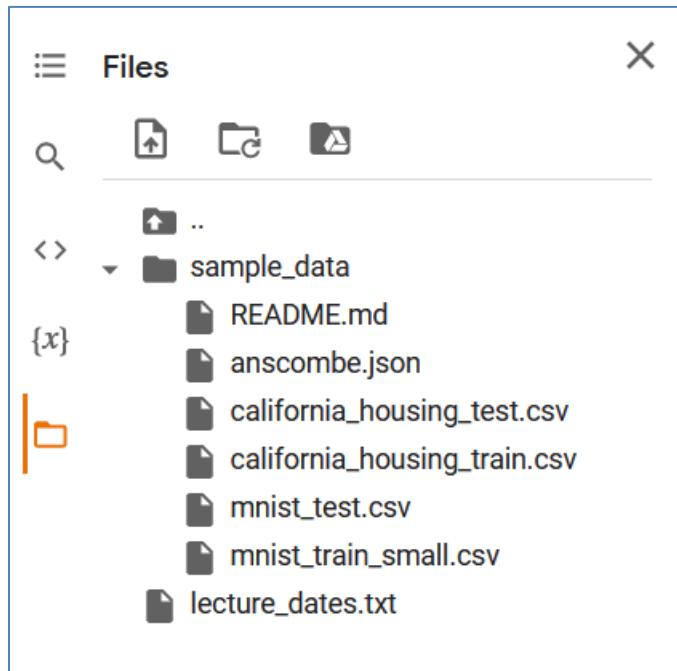
*os.remove(<file\_name>)*

```
import os  
os.remove("siber.txt")  
print("File Deleted Successfully...")  
  
File Deleted Successfully...
```

# Conceptualizing Python in Google COLAB

---

## Refresh Files



## Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The `os` module has several methods that help you create, remove, and change directories.

### The `mkdir()` Method

You can use the `mkdir()` method of the `os` module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created. The syntax of the method is shown below:

Syntax:

`os.mkdir(<dir_name>)`

# Conceptualizing Python in Google COLAB

## Creating Directories

```
import os  
os.mkdir("python")  
print("Directofy Created Successfully...")  
  
Directofy Created Successfully...
```

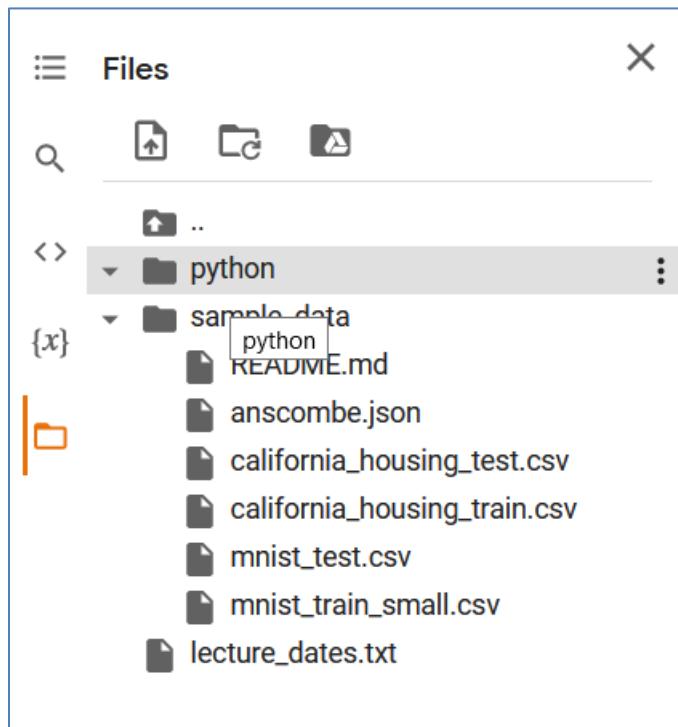
## The chdir() Method

You can use the chdir() method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory. The syntax of the method is shown below:

Syntax:

*os.chdir(<dir\_name>)*

Refresh Files



# Conceptualizing Python in Google COLAB

---

## Printing Current Working Directory

### The getcwd() Method

The getcwd() method displays the current working directory. The syntax of the method is shown below:

Syntax:

*os.getcwd()*

```
import os  
print(os.getcwd())  
  
/content
```

## Removing a Directory

### The rmdir() Method

The rmdir() method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed. The syntax of the method is shown below:

Syntax:

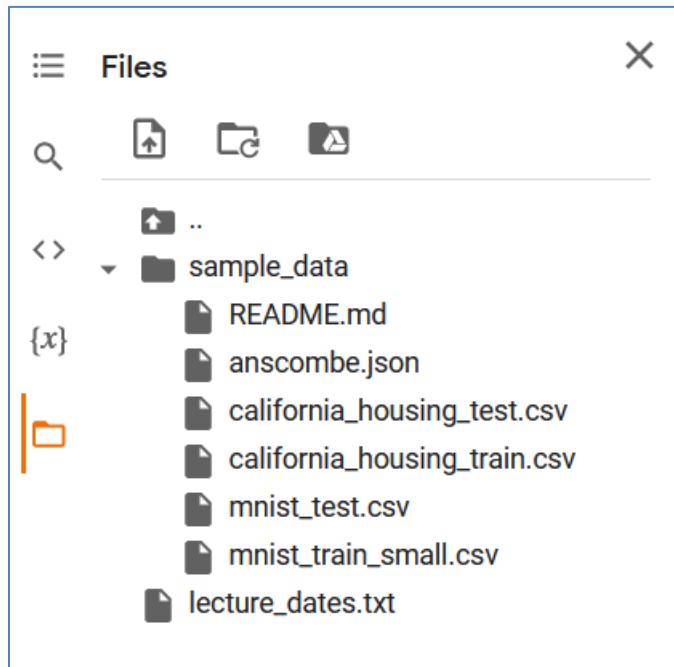
*os.rmdir(<dir\_name>)*

```
import os  
os.rmdir("python")  
print("Folder 'python' deleted successfully...")  
  
Folder 'python' deleted successfully...
```

# Conceptualizing Python in Google COLAB

---

## Refresh Files



# Conceptualizing Python in Google COLAB

---

## Chapter 10

### Lab Assignment on Regular Expressions in Python

Level – Intermediate

#### Regular Expression

A regular expression is a sequence of characters and meta characters that define a search pattern, mainly for use in pattern matching with strings.

#### Applications of Regular Expressions

Regular expression have wide applications in,

- Pattern matching
- Data validation
- Text searching
- URL matching etc.

#### ‘re’ Module in Python

Python has a module named ‘re’ to work with RegEx. Some important functions available in ‘re’ module are listed below:

##### re.match() Function

re.match() function is used to search pattern within the test\_string. It accepts two parameters:

- The first parameter is the regular expression pattern and
- The second parameter is the string to be tested.

The method returns a match object if the search is successful. If not, it returns None.

##### re.findall()

The re.findall() method returns a list of strings containing all matches.

# Conceptualizing Python in Google COLAB

---

Example 1: re.findall()

If the pattern is not found, re.findall() returns an empty list.

## Finding Multiple Occurrences of a Pattern

```
import re
string='hello 12 hi 89 howdy 34'
pattern='\d+'
result=re.findall(pattern,string)
print(result)
print(type(result))

['12', '89', '34']
<class 'list'>
```

## re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

If the pattern is not found, re.split() returns a list containing the original string.

```
import re
string='one:1 two:2 three:3 four:4'
pattern='\d+'
result=re.split(pattern,string)
print(result)

['one:', ' two:', ' three:', ' four:', '']
```

## re.sub()

The syntax of re.sub() is:

re.sub(pattern, replace, string)

# Conceptualizing Python in Google COLAB

---

The method returns a string where matched occurrences are replaced with the content of replace variable.

If the pattern is not found, re.sub() returns the original string.

## Replacing Multiple Spaces with a Single Space

In the following program, multiple white space characters are replaced with a single space using re.sub() method.

```
import re
string='SIBER           IS AN      AUTONOMOUS   INSTITUTE'
pattern='\s+'
replace=' '
result=re.sub(pattern,replace,string)
print(result)

SIBER IS AN AUTONOMOUS INSTITUTE
```

## Removing Spaces from a String

In the following program, all spaces within the string are removed using re.sub() method.

```
import re
string='SIBER IS AN AUTONOMOUS INSTITUTE'
pattern='\s+'
replace=''
result=re.sub(pattern,replace,string)
print(result)

SIBERISANAUTONOMOUSINSTITUTE
```

## re.subn()

The re.subn() is similar to re.sub() expect it returns a tuple of 2 items containing the new string and the number of substitutions made.

In the above program, if the statement re.sub() is replaced with re.subn(), a tuple containing the result string and the no. of occurrences is returned as shown in the following program:

# Conceptualizing Python in Google COLAB

```
import re
string='SIBER      IS  AN    AUTONOMOUS           INSTITUTE'
pattern='\s+'
replace=' '
result=re.subn(pattern,replace,string)
print(result)

('SIBER IS AN AUTONOMOUS INSTITUTE', 4)
```

The following program queries the return type of re.subn() method.

```
import re
string='SIBER      IS  AN    AUTONOMOUS           INSTITUTE'
pattern='\s+'
replace=' '
result=re.subn(pattern,replace,string)
print(result)
print(type(result))

('SIBER IS AN AUTONOMOUS INSTITUTE', 4)
<class 'tuple'>
```

You can pass count as a fourth parameter to the re.sub() method. If omitted, it results to 0. This will replace all occurrences. The following program replaces only the first occurrence of the multiple white spaces within the string with a single space.

```
import re
string='SIBER      IS  AN    AUTONOMOUS           INSTITUTE'
pattern='\s+'
replace=' '
result=re.subn(pattern,replace,string,1)
print(result)
print(type(result))

('SIBER IS  AN    AUTONOMOUS           INSTITUTE', 1)
<class 'tuple'>
```

# Conceptualizing Python in Google COLAB

---

## re.search()

The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, re.search() returns a match object; if not, it returns None.

***match = re.search(pattern, str)***

**Example:**

Here, the variable ‘found’ contains a match object. In the following program, the string ‘SIBER’ is searched in a given string. Since the search is successful, re.Match object is returned as depicted in the following figure:

```
import re
string='SIBER IS AN AUTONOMOUS INSTITUTE'
sstring='SIBER'
found=re.search(sstring,string)
if found:
    print("Search string found")
else:
    print("Search string not found")
print(type(found))

Search string found
<class 're.Match'>
```

Since the string ‘CSIBER’ is not found in the given string, re.search() method returns ‘NoneType’ as demonstrated in the following program:

```
import re
string='SIBER IS AN AUTONOMOUS INSTITUTE'
sstring='CSIBER'
found=re.search(sstring,string)
if found:
    print("Search string found")
else:
    print("Search string not found")
print(type(found))

Search string not found
<class 'NoneType'>
```

# Conceptualizing Python in Google COLAB

---

## Match object

You can get methods and attributes of a match object using dir() function as shown in the following program:

```
▶ import re  
print(dir(re.Match))  
['__class__', '__copy__', '__deepcopy__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'group', 'groups', 'groupdict']
```

Some of the commonly used methods and attributes of match objects are:

match.group()

The group() method returns the part of the string where there is a match.

```
▶ import re  
string='123-45 777-88'  
pattern='(\d{3})-(\d{2})'  
match=re.search(pattern, string)  
if match:  
    print(match.group())  
else:  
    print("Match not found")
```

```
123-45
```

Here, match variable contains a match object.

Our pattern  $(\d\{3\})(\d\{2\})$  has two subgroups  $(\d\{3\})$  and  $(\d\{2\})$ . You can get the part of the string of these parenthesized subgroups. Here's how:

# Conceptualizing Python in Google COLAB

```
import re
string='123-45 777-88'
pattern='(\d{3})-(\d{2})'
match=re.search(pattern, string)
print(match.group(1))
print(match.group(2))
print(match.group(1, 2))
print(match.groups())
```

```
123
45
('123', '45')
('123', '45')
```

match.start(), match.end() and match.span()

The start() function returns the index of the start of the matched substring. Similarly, end() returns the end index of the matched substring.

```
import re
string='123-45 777-88'
pattern='(\d{3})-(\d{2})'
match=re.search(pattern, string)
print(match.start())
print(match.end())
```

```
0
6
```

The span() function returns a tuple containing start and end index of the matched part.

```
import re
string='123-45 777-88'
pattern='(\d{3})-(\d{2})'
match=re.search(pattern, string)
print(match.span())
print(type(match.span()))
```

```
(0, 6)
<class 'tuple'>
```

# Conceptualizing Python in Google COLAB

---

## Using r prefix before RegEx

When r or R prefix is used before a regular expression, it means raw string. For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n.

Backslash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

### Example:

Raw string using r prefix

```
▶ import re
string='Python is easy\n Python is interesting \n Python is open source'
match=re.findall(r"\n", string)
print(match)
lines=len(match)+1
print("No. of Lines is a string : ",lines)

['\n', '\n']
No. of Lines is a string :  3
```

## match.re and match.string

The re attribute of a matched object returns a regular expression object. Similarly, string attribute returns the passed string.

```
▶ import re
string='123-45 777-88'
pattern='(\d{3})-(\d{2})'
match=re.search(pattern,string)
print(match.re)
print(match.string)

re.compile('(\d{3})-(\d{2})')
123-45 777-88
```

# Conceptualizing Python in Google COLAB

---

## MetaCharacters

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of commonly used metacharacters:

[] . ^ \$ \* + ? {} () \ |

## Searching a Pattern

### Met character . (Single Character)

```
import re
pattern='^a...ss'
test_string='abyss'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

```
import re
pattern='^a...s$'
test_string='ass'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Unsuccessful..

Replace pattern as shown below and re-execute:

pattern=':\d+'

# Conceptualizing Python in Google COLAB

---

```
import re
string='one:1 two:2 three:3 four:4'
pattern=':\d+'
result=re.split(pattern,string)
print(result)

['one', ' two', ' three', ' four', '']
```

## Character Repetitions

The following characters are employed to specify the repetition of characters in a string:

Meta character	Meaning
*	Zero or more occurrence of a preceding character
+	One or more occurrence of a preceding character
?	Zero or One occurrence of a character
.	Exactly one character

### \* Metacharacter (Zero or More Occurrence)

The pattern ‘c\*’ matches infinite no. of patterns, the strings containing ‘’, ‘c’, ‘cc’, ‘ccc’, . . . etc.

The following program demonstrates testing whether the given string matches the pattern ‘c\*’

The regular expression ab+c will give ac, abc, abbc, abbac, . . . and so on.

```
import re
pattern='c*'
test_string='csiber'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")

Search Successful..
```

# Conceptualizing Python in Google COLAB

---

```
import re
pattern='c*'
test_string='c'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

## ? Metacharacter (Zero or One Occurrence)

The pattern ‘c?’ matches two search patterns, the strings containing ‘’ and ‘c’. The following program demonstrates testing whether the given string matches the pattern ‘c?’

```
import re
pattern='c?'
test_string='csiber'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

The search pattern ‘docx?’ (x is optional) matches one of the patterns ‘doc’ or ‘docx’.

# Conceptualizing Python in Google COLAB

---

```
import re
pattern='docx?'
test_string='doc'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

```
import re
pattern='docx?'
test_string='docx'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

## + Metacharacter (One or More Occurrence)

The pattern ‘c+’ matches infinite no. of patterns, the strings containing ‘c’, ‘cc’, ‘ccc’, . . . etc. The following program demonstrates testing whether the given string matches the pattern ‘c+’

The relationship between the meta characters ‘\*’ and ‘+’ is that:

$$\{*\} = \{\} \cup \{+\}$$

Hence ‘c\*’ pattern contains all strings of ‘c+’ in addition to ‘ ’.

The regular expression ab+c will give abc, abbc, abbabc, . . . and so on.

# Conceptualizing Python in Google COLAB

---

```
import re
pattern='c+'
test_string='c'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

```
import re
pattern='c+'
test_string=''
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Unsuccessful..

```
import re
pattern='c*'
test_string=''
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

The curly braces {...}:

It informs to repeat the preceding character (or set of characters) for as many times as the value inside this bracket.

# Conceptualizing Python in Google COLAB

---

Example :

{2} means that the preceding character is to be repeated 2 times, {min,} means the preceding character is matches min or more times. {min, max} means that the preceding character is repeated at least min & at most max times.

## Character Classes

A character class matches any one of a set of characters. It is used to match the most basic element of a language like a letter, a digit, space, a symbol etc. The following table depicts different character classes which can be used in a regular expression:

Character Class	Meaning
/s	matches any whitespace characters such as space and tab
/S	matches any non-whitespace characters
/d	matches any digit character
/D	matches any non-digit characters
/w	matches any word character (basically alpha-numeric)
/W	matches any non-word character
/b	matches any word boundary (this would include spaces, dashes, commas, semi-colons, etc.)

## Setting Position for Match

^ (Beginning of the String)

The ^ symbol tells the computer that the match must start at the beginning of the string or line. The pattern '^d{3}' is interpreted as exactly 3 digits at the beginning of the string.

# Conceptualizing Python in Google COLAB

```
import re
pattern='^\d{3}'
test_string='012Employee'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

```
import re
pattern='^\d{3}'
test_string='01Employee'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Unsuccessful..

```
import re
pattern='^\d{3}'
test_string='0123Employee'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

## \$ (End of the String)

The \$ symbol tells the computer that the match must occur at the end of the string or before \n at the end of the line or string. The pattern '-\d{3}\$' is interpreted as the character '-' followed by exactly 3 digits at the end of the string.

# Conceptualizing Python in Google COLAB

---

```
import re
pattern='-\d{3}$'
test_string='-456'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

```
import re
pattern='-\d{3}$'
test_string='-456-'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Unsuccessful..

## Character Classes [] – Set of Characters

[set\_of\_characters] – Matches any single character in set\_of\_characters. By default, the match is case-sensitive.

### Example :

[abc] will match characters a, b and c in any string.

[^set\_of\_characters] – Negation: Matches any single character that is not in set\_of\_characters. By default, the match is case sensitive.

### Example :

[^abc] will match any character except a, b, c .

# Conceptualizing Python in Google COLAB

---

[first-last] – Character range: Matches any single character in the range from first to last.

## Example :

[a-zA-Z] will match any character from a to z or A to Z.

You can also specify a range of characters using - inside square brackets.

[a-e] is the same as [abcde].

[1-4] is the same as [1234].

[0-39] is the same as [01239].

## The Escape Symbol : \

If you want to match for the actual ‘+’, ‘.’ etc. characters literally and no with a special meaning, add a backslash(\) before that character. This will tell the computer to treat the following character as a search character and consider it for matching pattern as demonstrated in the following program. In the following program, the characters ‘+’ and ‘\*’ are interpreted literally and not as meta characters.

```
import re
pattern='\\d+[\\+-x\\*]\\d+'
test_string='2+2'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")

Search Successful..
```

# Conceptualizing Python in Google COLAB

---

```
import re
pattern='\\d+[+-x]*\\d+'
test_string='3*9'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")

Search Successful..
```

## Grouping Characters

A set of different symbols of a regular expression can be grouped together to act as a single unit and behave as a block, for this, you need to wrap the regular expression in the parenthesis( ).

### Example :

([A-Z]\w?) contains two different elements of the regular expression combined together. This expression will match any pattern containing uppercase letter followed by any optional character. (w stands for word character, alphanumeric)

Without parentheses the pattern

[A-Z]\w?

stands for

‘Any one character in the range A to Z followed by an optional white space character’

### Difference Between a(bc)\* and abc\*

The pattern a(bc)\* matches the strings, a, abc, abcabc, abcabcabc, . . . while the pattern abc\* matches the strings ab, abc, abcc, abccc, . . . etc.

### Alternation ( | ) :

Matches any one element separated by the vertical bar (|) character.

# Conceptualizing Python in Google COLAB

---

Example :

th(e|is|at) will match words - the, this and that.

```
import re
pattern='th(e|is|at)'
test_string='the'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")

Search Successful..
```

```
import re
pattern='th(e|is|at)'
test_string='this'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")

Search Successful..
```

```
import re
pattern='th(e|is|at)'
test_string='that'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")

Search Successful..
```

# Conceptualizing Python in Google COLAB

---

\number :

Backreference:

It allows a previously matched sub-expression(expression captured or enclosed within circular brackets ) to be identified subsequently in the same regular expression. \n means that group enclosed within the n<sup>th</sup> bracket will be repeated at current position.

Example :

([a-z])\1 will match “ee” in Seek because the character at second position is same as character at position 1 of the match.

Examples:

Regular Expression	Meaning
abc*	matches a string that has ab followed by zero or more c
abc+	matches a string that has ab followed by one or more c
abc?	matches a string that has ab followed by zero or one c
abc{2}	matches a string that has ab followed by 2 c
abc{2,}	matches a string that has ab followed by 2 or more c
abc{2,5}	matches a string that has ab followed by 2 up to 5 c
a(bc)*	matches a string that has a followed by zero or more copies of the sequence bc
a(bc){2,5}	matches a string that has a followed by 2 up to 5 copies of the sequence bc

Bracket expressions—[]

Regular Expression	Meaning
[abc]	matches a string that has either a or b or c and is the same as a b c
[a-c]	matches a string that has either a or b or c
[a-fA-F0-9]	a string that represents a single hexadecimal digit, case insensitively
[0-9]%	a string that has a character from 0 to 9 before a % sign
[^a-zA-Z]	a string that has not a letter from a to z or from A to Z. In this case the ^ is used as negation of the expression

# Conceptualizing Python in Google COLAB

---

## Look-ahead and Look-behind — (?=) and (?<=)

d(?=r) matches a d only if is followed by r, but r will not be part of the overall regex match

(?<=r)d matches a d only if is preceded by an r, but r will not be part of the overall regex match

You can use also the negation operator!

d(?!r) matches a d only if is not followed by r, but r will not be part of the overall regex match

(?<!r)d matches a d only if is not preceded by an r, but r will not be part of the overall regex match

## Validate Hexadecimal No. Using Regular Expression

```
import re
pattern='^[A-Fa-f0-9]*$'
test_string='0A1B'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..

```
import re
pattern='^[A-Fa-f0-9]*$'
test_string='0A1Z'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Unsuccessful..

# Conceptualizing Python in Google COLAB

---

## Character Validation



```
import re
pattern='^A-Za-z*$'
test_string='MCA'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Successful..



```
import re
pattern='^A-Za-z*$'
test_string='MCA1'
result=re.match(pattern,test_string)
if result:
    print("Search Successful..")
else:
    print("Search Unsuccessful..")
```

Search Unsuccessful..

## Exercise on Regular Expressions

Give regular expressions for the following

- i. non-digit character `[^0-9]`
- ii. first letter of a word is capital `[A-Z]\w+`
- iii. All words starting with ab `^ab.*`
- iv. All words starting with ab and ending with cd `^ab.*cd$`
- v. Integer with any no. of digits `\d+`

# Conceptualizing Python in Google COLAB

---

Give the meaning of following patterns:

**`^a...b$`**

A string containing exactly five characters starting with ‘a’, ending with ‘b’ and any three characters between ‘a’ and ‘b’.

**`[0-9]{2, 4}`**

It matches at least 2 digits but not more than 4 digits

**`^(a-zA-Z0-9_-|.)+@[a-zA-Z0-9_-|.]++\.(a-zA-Z){2,5}$`**

Regular expression for an email address :

## Chapter 11

### Lab Assignment on Language Basics and Error Handling in Python

Level – Intermediate

#### Program 1: Scope of Variables

##### Local Scope

A variable declared within a function is local to the function in which it is declared. In the following program ‘x’ is used in different scopes:

- Global scope
- Local function scope
- Parameter to a function

Even if variable name is ‘x’ its scope is different. In the following program, x is declared as a global variable which is initialized to 50. The variable with the same name is declared inside a function func(). However, its scope is limited to a function in which it is declared. When the function returns the variable goes out of scope.

The screenshot shows a Google Colab interface. On the left, there is a code cell containing Python code. On the right, the output of the code is displayed. The code defines a global variable `x = 50`, then defines a function `func(x)` that prints the value of `x` and changes its value to 2. It then calls `func(x)` and prints the value of `x` again. The output shows the expected behavior: the global `x` is printed as 50, the local `x` is changed to 2, and the global `x` is then printed again as 50.

```
x = 50
def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)
func(x)
print('x is now', x)
```

```
x is 50
Changed local x to 2
x is now 50
```

# Conceptualizing Python in Google COLAB

## Global Scope

The variable declared outside any function attains a global scope and is shared between all functions in the same program. When a function has a local variable with the same name as global variable, the local variable takes preference. To access a global variable within a function, use ‘**global**’ keyword as demonstrated in the following program:

```
x = 50
def func():
    global x
    print('x is', x)
    x = 2
    print('Changed global x to', x)
func()
print('Value of x is', x)
```

x is 50  
Changed global x to 2  
Value of x is 2

## Exercise 1:

What error will occur when you execute the following code?

MANGO = APPLE

On execution of the above program, ‘**NameError**’ with the message ‘**name APPLE is not defined**’ is displayed as shown in the following program since in Python a variable cannot be used without initializing it.

```
MANGO=APPLE
```

---

```
NameError                                 Traceback (most recent call last)
<ipython-input-15-ccal03bec844> in <module>()
      1 MANGO=APPLE

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

SEARCH STACK OVERFLOW

# Conceptualizing Python in Google COLAB

---

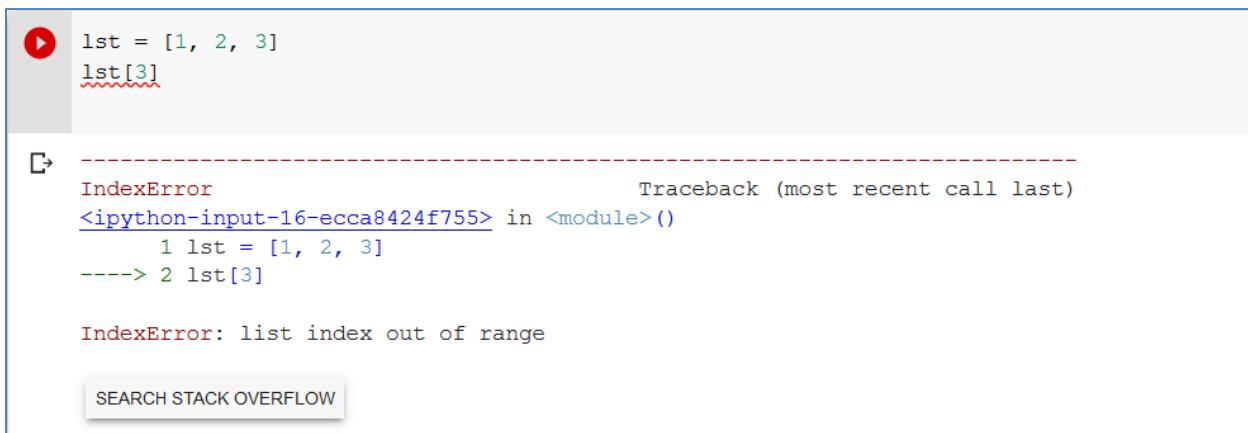
## Exercise 2:

What will be the output of the following Python code?

```
lst = [1, 2, 3]
```

```
lst[3]
```

On execution of the above code '**IndexError**' with the message '*list index out of range*' is displayed as shown in the following figure. Python performs bounds checking on data structures such as list, tuple etc.



The screenshot shows a Jupyter Notebook cell with the following code:

```
lst = [1, 2, 3]
lst[3]
```

Below the code, a red play button icon is visible. A dashed line indicates the start of the traceback:

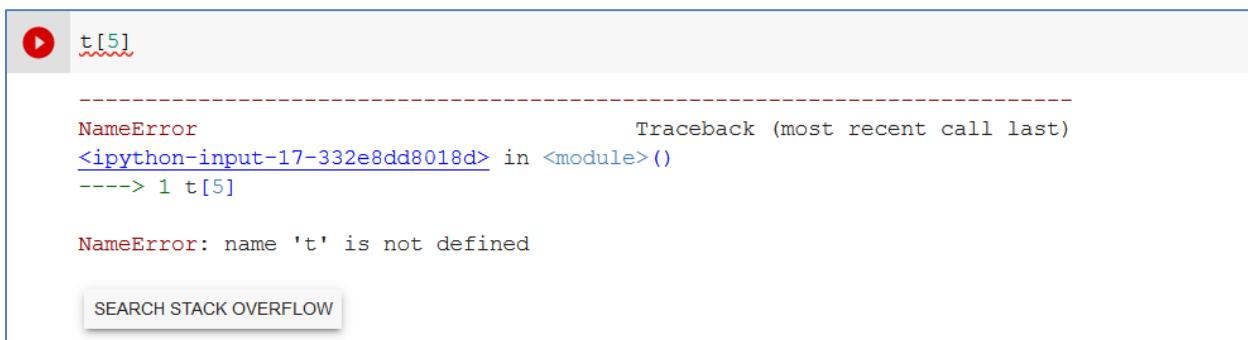
```
-----  
IndexError                                     Traceback (most recent call last)  
<ipython-input-16-ecca8424f755> in <module>()  
      1 lst = [1, 2, 3]  
----> 2 lst[3]  
  
IndexError: list index out of range
```

A 'SEARCH STACK OVERFLOW' button is located at the bottom of the cell.

## Exercise 3:

What will be the output of the following Python code?

```
t[5]
```



The screenshot shows a Jupyter Notebook cell with the following code:

```
t[5]
```

Below the code, a red play button icon is visible. A dashed line indicates the start of the traceback:

```
-----  
NameError                                     Traceback (most recent call last)  
<ipython-input-17-332e8dd8018d> in <module>()  
----> 1 t[5]  
  
NameError: name 't' is not defined
```

A 'SEARCH STACK OVERFLOW' button is located at the bottom of the cell.

# Conceptualizing Python in Google COLAB

On declaring a list with the name ‘t’ containing three elements, ‘**IndexError**’ exception is raised with the message ‘**list index out of range**’ as shown in the following figure:

The screenshot shows a Google Colab cell with the following code:

```
t=[1,2,3]
t[5]
```

Below the code, a red error icon is visible. The output shows a stack trace and the error message:

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-2-1202e68c67f9> in <module>()
      1 t=[1,2,3]
----> 2 t[5]

IndexError: list index out of range
```

At the bottom of the cell, there is a button labeled "SEARCH STACK OVERFLOW".

The program executes successfully, on adding three more elements to the list as shown below:

The screenshot shows a Google Colab cell with the following code:

```
t=[1,2,3,4,5,6]
t[5]
```

The output shows the value 6.

## Exercise 4:

What will be the output of the following Python code, if the time module has already been imported?

4 + '3'

On execution of the above statement, ‘**TypeError**’ is generated with the message ‘**unsupported operand types**’ as shown in the following figure:

The screenshot shows a Google Colab cell with the following code:

```
4 + '3'
```

Below the code, a red error icon is visible. The output shows a stack trace and the error message:

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-18-23fdc3378411> in <module>()
----> 1 4 + '3'

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

At the bottom of the cell, there is a button labeled "SEARCH STACK OVERFLOW".

# Conceptualizing Python in Google COLAB

---

## Exercise 5:

What will be the output of the following Python code?

```
int('65.43')
```

On execution of the above program ‘*ValueError*’ is generated with the message ‘*Invalid literal for int()*’ as shown in the following figure:

The screenshot shows a code cell in Google Colab. The user has typed `int('65.43')`. A red error icon is visible next to the cell. The output shows a `ValueError` traceback:

```
int('65.43')

-----
ValueError                                Traceback (most recent call last)
<ipython-input-19-d45f4a0303e0> in <module>()
      1 int('65.43')

ValueError: invalid literal for int() with base 10: '65.43'

SEARCH STACK OVERFLOW
```

## Exercise 6:

What will be the output of the following Python code?

```
student={"rollno":1,"name":"Maya"}  
print(student["division"])
```

On execution of the above program, ‘*KeyError: division*’ error is generated since in the ‘student’ dictionary there is no key with the name ‘division’.

## KeyError

The screenshot shows a code cell in Google Colab. The user has typed `student={"rollno":1, "name":"Maya"} print(student["division"])`. A red error icon is visible next to the cell. The output shows a `KeyError` traceback:

```
student={"rollno":1, "name":"Maya"}  
print(student["division"])

-----
KeyError                                Traceback (most recent call last)
<ipython-input-21-fb7e3158c987> in <module>()
      1 student={"rollno":1, "name":"Maya"}
      2 print(student["division"])

KeyError: 'division'

SEARCH STACK OVERFLOW
```

# Conceptualizing Python in Google COLAB

---

## IndentationError

Python uses indentation for defining a block of code referred to as '**Suite**'. The indentation error occurs when the spaces or tabs are not placed properly.

```
▶ student={"rollno":1, "name":"Maya"}  
~~~~~ print(student["rollno"])  
  
File "<ipython-input-23-5280d887c348>", line 2  
    print(student["rollno"])  
    ^  
IndentationError: unexpected indent
```

[SEARCH STACK OVERFLOW](#)

# Conceptualizing Python in Google COLAB

---

## Chapter 12

### Lab Assignment on Object Oriented Programming in Python

#### Level – Intermediate

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods.

#### Class and Object

A class is a template or a blue print for creating objects. An object is a class variable or a runtime instance of a class.

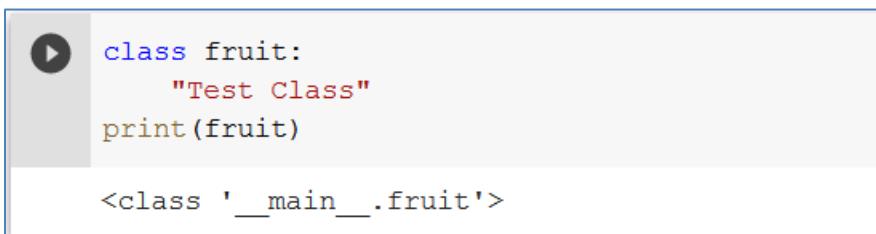
#### Creating Classes in Python

The *class* statement is used for creating a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows – The syntax for creating a class in Python is shown below:

```
class ClassName:  
    'Optional class documentation string'  
    class_suite
```

The class has a documentation string, which can be accessed via *ClassName.\_\_doc\_\_*.

The *class\_suite* consists of all the component statements defining class members, data attributes and functions.



The screenshot shows a Google Colab interface. On the left, there is a play button icon. The code cell contains the following Python code:

```
class fruit:  
    "Test Class"  
    print(fruit)
```

The output of the code cell is:

```
<class '__main__.fruit'>
```

# Conceptualizing Python in Google COLAB

---

## Displaying Document String

Every Python class has a member `__doc__()` which can be used for accessing the document string of a class as shown in the following program:

```
class fruit:  
    "Test Class"  
print(fruit)  
print(fruit.__doc__)  
  
<class '__main__.fruit'>  
Test Class
```

## ‘self’ Keyword in Python

The ‘`self`’ keyword is used for referring to the data member of the class. Without ‘`self`’ keyword, the variable is considered to be the local variable. Also, every member function of the class must have ‘`self`’ as its first parameter as shown in the following program. However ‘`self`’ is not used while invoking a method.

```
class fruit:  
    "Test Class"  
    def sayhi(self):  
        print("Hi")  
f=fruit()  
f.sayhi()  
fruit.sayhi  
  
Hi  
<function __main__.fruit.sayhi>
```

## Object Creation in Python

`__new__` is a static class method that lets us control object creation. Whenever we make a call to the class constructor, it makes a call to `__new__`. Internally, `__new__` is the constructor that returns a valid and unpopulated object on which to call `__init__`.

# Conceptualizing Python in Google COLAB

---

## Constructors

`__init__()` method is used for defining a constructor in Python which takes '`self`' as its first argument. Python rebinds the name `__init__` to the `__new__` method. This means in the following program, the first declaration of this method is inaccessible now.

```
▶ class Person:  
    def __init__(self):  
        print("I am first constructor")  
  
    def __init__(self):  
        print("I am second constructor")  
  
p1 = Person()  
  
I am second constructor
```

## Class Instantiation

To create instances of a class, you call the class using class name and pass in whatever arguments its `__init__` method accepts. The syntax for instantiating an object of the class is shown below:

Syntax:

`<instance_name> = <class_name>(<parameter_list>)`

# Conceptualizing Python in Google COLAB

```
▶ class Person:  
    def __init__(self):  
        print("I am first constructor")  
  
    def __init__(self, name):  
        print(self.name)  
        print("I am second constructor")  
  
p1 = Person("MCA")  
  
-----  
AttributeError Traceback (most recent call last)  
<ipython-input-8-2357c96d7b16> in <module>()  
      8  
      9  
---> 10 p1 = Person("MCA")  
  
<ipython-input-8-2357c96d7b16> in __init__(self, name)  
    4  
    5     def __init__(self, name):  
----> 6         print(self.name)  
    7         print("I am second constructor")  
    8  
  
AttributeError: 'Person' object has no attribute 'name'  
  
SEARCH STACK OVERFLOW
```

In the above program, the data member with the name '**name**' is not defined. The bug is fixed in the following program. The '**name**' parameter passed to the class constructor is used for initializing the data member '**name**' of the class.

```
▶ class Person:  
    def __init__(self):  
        print("I am first constructor")  
  
    def __init__(self, name):  
        self.name=name  
        print(self.name)  
  
p1 = Person("MCA")  
  
◀ MCA
```

# Conceptualizing Python in Google COLAB

---

In the following program, ‘*empCount*’ is a static variable which is shared by all instances of ‘*employee*’ class. However, ‘*name*’ and ‘*salary*’ are data members or instance members of the class. The class has a two argument constructor `__init__()` for initializing the data members of a class. The class has two member functions `displayCount()` and `displayEmployee()` for displaying the total count of employees and employee information.

```
class Employee:  
    'Common base class for all employees'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print("Total Employee %d" % Employee.empCount)  
  
    def displayEmployee(self):  
        print("Name : ", self.name, " , Salary: ", self.salary)  
  
emp1 = Employee("Zara", 2000)  
emp2 = Employee("Manni", 5000)  
emp1.displayEmployee()  
emp2.displayEmployee()  
print("Total Employee %d" % Employee.empCount)  
  
Name : Zara , Salary: 2000  
Name : Manni , Salary: 5000  
Total Employee 2
```

**Note:** Within the class local variable is referenced directly while class variable is referenced using the following syntax (using class name):

*<class\_name>.<class\_variable\_name> and*

data members are referenced using the following syntax:

*self.<data\_member\_name> and*

There can be only one `__init__()` method in a class. If the class contains multiple `__init__()` methods, then the *last* `__init__()` method takes effect by overwriting earlier `__init__()` methods, if any as demonstrated in the following program. In the following program, the ‘Person’ class

# Conceptualizing Python in Google COLAB

contains three `__init__()` methods of which the last `__init__()` method will be considered for object creation. An attempt to create a '**Person**' object with a single '**name**' parameter generates '**TypeError**' with the message '**`__init__()` missing one required positional parameter 'age'**' as shown below:

```
▶ class Person:
    def __init__(self):
        name=""
        age=0
    def __init__(self, name):
        age=0
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)
print(p1.name)
print(p1.age)
p1 = Person(name="Jack")
print(p1.name)
print(p1.age)

□ John
36
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-5-6327588f3cc8> in <module>()
      12 print(p1.name)
      13 print(p1.age)
---> 14 p1 = Person(name="Jack")
      15 print(p1.name)
      16 print(p1.age)

TypeError: __init__() missing 1 required positional argument: 'age'

SEARCH STACK OVERFLOW
```

## Default Arguments

The constructor can accept default parameters. The constructor overloading can be simulated using default arguments in a constructor as shown in the following program. In the following example, the 'Person' class has two parameter constructor for initializing 'name' and 'age' data members which offers three different ways of creating the object of 'Person' class as described below:

### Method 1: Using default constructor

`p1 = Person()`

The above statement creates a 'Person' object with `name='xyz'` and `age=20`.

# Conceptualizing Python in Google COLAB

---

Method 2: Using one argument constructor

**p1 = Person(name="Jack")**

The above statement creates a ‘Person’ object with name=’Jack’ and age=20.

**p1 = Person(age=30)**

The above statement creates a ‘Person’ object with name=’xyz’ and age=30.

Method 3: Using two argument constructor

**p1 = Person(name="John",36)**

The above statement creates a ‘Person’ object with name=’John’ and age=36.

A screenshot of a Google Colab notebook. On the left, there is a code editor window containing Python code. On the right, there is an output window showing the results of the code execution.

**Code Editor Content:**

```
class Person:  
    def __init__(self, name="xyz", age=20):  
        self.name = name  
        self.age = age  
  
p1 = Person("John", 36)  
print(p1.name)  
print(p1.age)  
  
p1 = Person(name="Jack")  
print(p1.name)  
print(p1.age)  
  
p1 = Person()  
print(p1.name)  
print(p1.age)
```

**Output Window Content:**

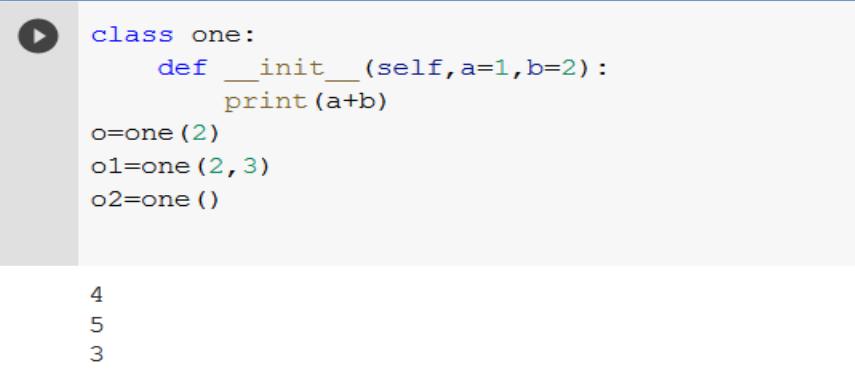
```
John  
36  
Jack  
20  
xyz  
20
```

## Constructor with Default Arguments

In the following program, the constructor of class ‘one’ takes two parameters ‘a’ and ‘b’ with the default values of 1 and 2, respectively.

# Conceptualizing Python in Google COLAB

---



```
class one:
    def __init__(self,a=1,b=2):
        print(a+b)
o=one(2)
o1=one(2,3)
o2=one()

4
5
3
```

## Destroying Objects (Garbage Collection)

The memory management in Python is automatic. Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python employs reference counting mechanism to figure out unneeded objects. Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes. An unreferenced object is referred to as '*garbage*'.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method *\_\_del\_\_()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

In the following program, the '*Point*' object is referenced by three references, '*pt1*', '*pt2*' and '*pt3*'.*getrefcount()* method of '*sys*' class is employed for displaying reference count which accepts a single parameter the name of the reference. Every time a new reference is created to the existing

# Conceptualizing Python in Google COLAB

‘**Point**’ class object, the reference count is incremented by 1 till it reaches 3. In the subsequent statements, the references ‘**pt1**’, ‘**pt2**’ and ‘**pt3**’ are deleted in that order. When no more references exist for ‘**Point**’ object it is garbage collected and prior to that `__del__()` method is invoked which displays the message ‘**Point Destroyed**’.

```
▶ import sys
class Point:
    def __init__( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
        print(class_name, "destroyed")

pt1 = Point()
print("Reference Count : ",sys.getrefcount(pt1)-1)
pt2 = pt1
print("Reference Count : ",sys.getrefcount(pt1)-1)
pt3 = pt1
print("Reference Count : ",sys.getrefcount(pt1)-1)
del pt3
print("Reference Count : ",sys.getrefcount(pt1)-1)
del pt2
print("Reference Count : ",sys.getrefcount(pt1)-1)
del pt1

Reference Count : 1
Reference Count : 2
Reference Count : 3
Reference Count : 2
Reference Count : 1
Point destroyed
```

## Class IDs

The `id()` function returns a unique id for the specified object. All objects in Python has its own unique id. The id is assigned to the object when it is created. In the following example, the ‘**Point**’ class has a constructor `__init__()` with two default arguments for initializing the x and y coordinates and a destructor `__del__()`. pt1, pt2 and pt3 are three references to the same ‘**Point**’ object. Hence the ‘`id`’ is same for all references as shown in the output generated on execution of the above program:

# Conceptualizing Python in Google COLAB

```
class Point:  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
    def __del__(self):  
        class_name = self.__class__.__name__  
        print(class_name, "destroyed")  
  
pt1 = Point()  
pt2 = pt1  
pt3 = pt1  
print(id(pt1), id(pt2), id(pt3)) # prints the ids of the objects  
del pt1  
del pt2  
del pt3
```

```
140374062186448 140374062186448 140374062186448  
Point destroyed
```

When ‘**del**’ function is used for deleting the object, the destructor **\_\_del\_\_()** is invoked.

```
class Point:  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
    def __del__(self):  
        class_name = self.__class__.__name__  
        print(class_name, "destroyed")  
  
pt1 = Point()  
pt2 = Point()  
pt3 = pt1  
print(id(pt1), id(pt2), id(pt3)) # prints the ids of the objects  
del pt1  
del pt2  
del pt3
```

```
↳ 140374061767696 140374061767824 140374061767696  
Point destroyed  
Point destroyed
```

## Accessing the Class Members

Instead of using the normal statements to access attributes, you can use the functions depicted in the following table:

# Conceptualizing Python in Google COLAB

Function Name	Description
getattr(obj, name[, default])	to access the attribute of object.  <b>Example:</b>  getattr(emp1, 'age')  Returns value of 'age' attribute of 'emp1' instance.
hasattr(obj, name)	to check if an attribute exists or not.  <b>Example:</b>  hasattr(emp1, 'age')  Returns true if 'age' attribute exists in class of which 'emp1' is an instance.
setattr(obj, name, value)	to set an attribute. If attribute does not exist, then it would be created.  <b>Example:</b>  setattr(emp1, 'age', 8)  Sets attribute 'age' of 'emp' instance to 8
delattr(obj, name)	to delete an attribute.  <b>Example:</b>  delattr(empl, 'age')  Delete attribute 'age' of class of which 'emp1' is an instance.

## Built-In Class Attributes

Every Python class keeps following built-in attributes which are depicted in the following table and they can be accessed using dot operator like any other attribute.

Attribute Name	Description
__dict__	Dictionary containing the class's namespace.
__doc__	Class documentation string or None, if undefined
__name__	Class name.

# Conceptualizing Python in Google COLAB

<code>__module__</code>	Module name in which the class is defined. This attribute is " <code>__main__</code> " in interactive mode.
<code>__bases__</code>	A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

The above attribute are accessed for the '**Employee**' class as shown in the following figure:

The screenshot shows a Python code cell in Google Colab. The code defines a class `Employee` with methods `__init__`, `displayCount`, and `displayEmployee`. It also prints various \_\_attribute\_\_ values. The output shows the class definition and the printed attribute values.

```
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print("Total Employee %d" % Employee.empCount)

    def displayEmployee(self):
        print("Name : ", self.name, ", Salary: ", self.salary)

print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)

Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>)
Employee.__dict__: {'__module__': '__main__', '__doc__': 'Common base class for all employees', 'empCount': 0, '__init__': <function Employee.__init__ at 0x7fab62154290'}
```

## Class Inheritance

Inheritance aids in code reusability. If two classes have common traits, then instead of creating the class from scratch, it can simply be inherited from another class. The class from which a new class is inherited is referred to as '**Base**' class or '**Super**' class and the new class created is referred to as '**Derived**' or '**Sub**' class. The '**Sub**' class can do one of the following:

- Inherit the members of base class
- Override the members of base class
- Add new data members of functionality to the class

The syntax for class inheritance is shown below:

### Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

# Conceptualizing Python in Google COLAB

---

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    'Optional class documentation string'  
    class_suite
```

More than one base class can be listed in the parentheses since Python supports multiple inheritance.

In the following example, ‘**Child**’ is the derived class of ‘**Parent**’ which inherits the members of ‘**Parent**’ class and adds a new member function with the name *childMethod()*

```
class Parent:      # define parent class  
    parentAttr = 100  
    def __init__(self):  
        print("Calling parent constructor")  
  
    def parentMethod(self):  
        print('Calling parent method')  
  
    def setattr(self, attr):  
        Parent.parentAttr = attr  
  
    def getattr(self):  
        print("Parent attribute :", Parent.parentAttr)  
  
class Child(Parent): # define child class  
    def __init__(self):  
        print("Calling child constructor")  
  
    def childMethod(self):  
        print('Calling child method')  
  
c = Child()      # instance of child  
c.childMethod()   # child calls its method  
c.parentMethod()  # calls parent's method  
c.setAttr(200)    # again call parent's method  
c.getAttr()       # again call parent's method
```

On execution of the above program, the following output is generated:

```
Calling child constructor  
Calling child method  
Calling parent method  
Parent attribute : 200
```

# Conceptualizing Python in Google COLAB

## Method Overriding

The child class can override the functionality of the parent class to define a different functionality in a derived class. In the following example, the *my Method()* method inherited from '**Parent**' class is overridden in '**Child**' class.

```
▶ class Parent:      # define parent class
    def myMethod(self):
        print('Calling parent method')

class Child(Parent): # define child class
    def myMethod(self):
        print('Calling child method')

c = Child()          # instance of child
c.myMethod()         # child calls overridden method

Calling child method
```

Modify the above program to initialize the variable '*c*' with the reference of '**Parent**' class and re-execute the program as shown below:

```
▶ class Parent:      # define parent class
    def myMethod(self):
        print('Calling parent method')

class Child(Parent): # define child class
    def myMethod(self):
        print('Calling child method')

c = Parent()          # instance of parent
c.myMethod()          # child calls overridden method

Calling parent method
```

# Conceptualizing Python in Google COLAB

---

## Access Modifiers in Python : Public, Private and Protected

Various object-oriented languages like C++, Java, Python controls access modifications which are used to restrict access to the variables and methods of the class. Most programming languages has three forms of access modifiers, which are **Public**, **Protected** and **Private** in a class. Python uses ‘\_’ symbol to determine the access control for a specific data member or a member function of a class. Access specifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.

A Class in Python has three types of access modifiers –

- Public Access Modifier
- Protected Access Modifier
- Private Access Modifier

### Public Access Modifier:

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

### A Program to Illustrate Public Access Modifier in a Class

# Conceptualizing Python in Google COLAB

```
class Student:

    # constructor
    def __init__(self, name, rollno):

        # public data members
        self.name = name
        self.rollno = rollno

    # public member function
    def displayRollno(self):
        # accessing public data member
        print("Rollno : ", self.rollno)

# creating object of the class
obj = Student("Milan", 1)

# accessing public data member
print("Name: ", obj.name)

# calling public member function of the class
obj.displayRollno()
```

```
Name: Milan
Rollno : 1
```

In the above program, ‘*name*’ and ‘*rollno*’ are public data members and *displayRollno()* method is a public member function of the class ‘*Student*’. These data members of the class ‘*Student*’ can be accessed from anywhere in the program.

## Protected Access Modifier:

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore ‘\_’ symbol before the data member of that class.

## Program to illustrate protected access modifier in a class

In the following program, *\_name*, *\_roll* and *\_elective* are protected data members and *\_displayRollAndElective()* method is a protected method of the super class *Student*. The *displayDetails()* method is a public member function of the class ‘*MCAStudent*’ which is derived from the ‘*Student*’ class, the *displayDetails()* method in ‘*MCAStudent*’ class accesses the protected data members of the *Student* class.

# Conceptualizing Python in Google COLAB

```
# super class
class Student:
    # protected data members
    _name = None
    _roll = None
    _elective = None

    # constructor
    def __init__(self, name, roll, elective):
        self._name = name
        self._roll = roll
        self._elective = elective

    # protected member function
    def _displayRollAndElective(self):

        # accessing protected data members
        print("Roll      : ", self._roll)
        print("Elective : ", self._elective)
```

```
class MCAStudent(Student):
    # constructor
    def __init__(self, name, roll, elective):
        Student.__init__(self, name, roll, elective)

    # public member function
    def displayDetails(self):

        # accessing protected data members of super class
        print("Name      : ", self._name)

        # accessing protected member functions of super class
        self._displayRollAndElective()

    # creating objects of the derived class
    obj = MCAStudent("Milan", 1, "Machine Learning with Python")

    # calling public member functions of the class
    obj.displayDetails()

Name      : Milan
Roll      : 1
Elective : Machine Learning with Python
```

## Private Access Modifier:

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore ‘\_\_’ symbol before the data member of that class.

# Conceptualizing Python in Google COLAB

---

In the following program, `_name`, `_roll` and `_elective` are private members, `__displayDetails()` method is a private member function (these can only be accessed within the class) and `accessPrivateFunction()` method is a public member function of the class '**Student**' which can be accessed from anywhere within the program. The `accessPrivateFunction()` method accesses the private members of the class '**Student**'.

```
class Student:
    # private members
    __name = None
    __roll = None
    __elective = None

    # constructor
    def __init__(self, name, roll, elective):
        self.__name = name
        self.__roll = roll
        self.__elective = elective

    # private member function
    def __displayDetails(self):

        # accessing private data members
        print("Name      : ", self.__name)
        print("RollNo   : ", self.__roll)
        print("Elective : ", self.__elective)

    # public member function
    def accessPrivateFunction(self):

        # accesing private member function
        self.__displayDetails()

# creating object
obj = Student("Milan", 1, "Android Development with Kotlin")

# calling public member function of the class
obj.accessPrivateFunction()
```

The following program illustrates the use of all the above three access modifiers (public, protected and private) of a class in Python:

In the following program, `x`, `_y` and `__z` are public, protected and private data members of '**Super**' class. The class has three argument constructor for initializing these data members. Also the class has the following member functions:

`display_public()` → for displaying a public data member, `x`

# Conceptualizing Python in Google COLAB

---

*display\_protected()* → for displaying a public data member, `_y`

*display\_private()* → for displaying a public data member, `__z`.

*access\_private()* → for accessing private data member of the class outside the class.

The ‘**Derived**’ class has a three argument constructor which invokes the super class constructor for initializing the data members of the class. The super class constructor can be invoked in one of the following ways:

**Method 1:** Using super class name

`<superclass_name>.__init__(self,x,y,z)`

`Super().__init__(self,x,y,z)`

‘self’ parameter should be the first parameter while using class name.

**Method 2:** Using super() method

`super().__init__(x,y,z)`

When using `super()`, ‘self’ is not used as a first parameter to the method call.

# Conceptualizing Python in Google COLAB

```
class Super:  
    x = None  
    _y = None  
    __z=None  
    def __init__(self, x, y, z):  
        self.x=x  
        self._y=y  
        self.__z=z  
    def display_public(self):  
        print("Public Member : ",self.x)  
    def display_protected(self):  
        print("Protected Member : ",self._y)  
    def display_private(self):  
        print("Private Member : ",self.__z)  
    def access_private(self):  
        self.display_private()  
  
class Derived(Super):  
    def __init__(self, x, y, z):  
        Super.__init__(self,x,y,z)  
    def access_protected(self):  
        self.display_protected()  
obj=Derived(10,20,30)  
obj.display_public()  
obj.access_protected()  
obj.access_private()
```

```
↳ Public Member : 10  
Protected Member : 20  
Private Member : 30
```

The above program can be re-written using super() function to access the super class functionality as shown below:

# Conceptualizing Python in Google COLAB

```
class Super:  
    x = None  
    _y = None  
    __z=None  
    def __init__(self, x, y, z):  
        self.x=x  
        self._y=y  
        self.__z=z  
    def display_public(self):  
        print("Public Member : ",self.x)  
    def display_protected(self):  
        print("Protected Member : ",self._y)  
    def display_private(self):  
        print("Private Member : ",self.__z)  
    def access_private(self):  
        self.display_private()  
  
class Derived(Super):  
    def __init__(self, x, y, z):  
        super().__init__(x,y,z)  
    def access_protected(self):  
        super().display_protected()  
obj=Derived(10,20,30)  
obj.display_public()  
obj.access_protected()  
obj.access_private()
```

```
↳ Public Member : 10  
Protected Member : 20  
Private Member : 30
```

The '**Derived**' class has direct access to the members x and \_y.

```
obj=Derived(10,20,30)  
print("Public Member : ",obj.x)  
print("Protected Member : ",obj._y)  
obj.display_private()
```

```
↳ Public Member : 10  
Protected Member : 20  
Private Member : 30
```

# Conceptualizing Python in Google COLAB

```
obj=Super(10,20,30)
print("Public Member      : ",obj.x)
print("Protected Member   : ",obj._y)
#print("Private Member    : ",obj.__z)
obj.display_private()
```

```
↳ Public Member      : 10
    Protected Member   : 20
    Private Member    : 30
```

An attempt to access the private data member outside the class generates '*AttributeError*' as shown in the following program. The protected members are still accessible in Python.

```
obj=Super(10,20,30)
print("Public Member      : ",obj.x)
print("Protected Member   : ",obj._y)
print("Private Member    : ",obj.__z)
#obj.display_private()
```

```
↳ Public Member      : 10
    Protected Member   : 20
-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-25-de8c65643508> in <module>()
    24 print("Public Member      : ",obj.x)
    25 print("Protected Member   : ",obj._y)
--> 26 print("Private Member    : ",obj.__z)
    27 #obj.display_private()
    28

AttributeError: 'Super' object has no attribute '__z'
```

SEARCH STACK OVERFLOW

## issubclass() and isinstance() Methods

You can use *issubclass()* or *isinstance()* functions to check a relationships of two classes and instances.

- The *issubclass(sub, sup)* boolean function returns true if the given subclass '*sub*' is indeed a subclass of the superclass '*sup*'.
- The *isinstance(obj, Class)* boolean function returns true if '*obj*' is an instance of class '*Class*' or is an instance of a subclass of Class

# Conceptualizing Python in Google COLAB

---

These methods are demonstrated in the following program:

```
class Parent:  
    def myMethod(self):  
        print('Calling parent method')  
  
class Child(Parent):  
    def myMethod(self):  
        print('Calling child method')  
  
c = Child()  
print(issubclass(Child, Parent))  
print(issubclass(Parent, Child))  
  
True  
False
```

## isinstance() Method Demo

The following program demonstrates the use of *isinstance()* method. The *isinstance()* method returns '**True**' if the instance passed to it as first argument is an instance of the class passed to the method as second argument and all other classes which are above it in class hierarchy. In the following program, the variable 'c' is an instance of 'Child' class which is a derived class of 'Parent' class. Hence both the following statements return 'True'.

*isinstance(c, Parent)*

*isinstance(c, Child)*

However, the statement

*isinstance(c, Student)* returns 'False' since 'Student' and 'Child' are unrelated.

# Conceptualizing Python in Google COLAB

```
class Parent:  
    def myMethod(self):  
        print('Calling parent method')  
  
class Child(Parent):  
    def myMethod(self):  
        print('Calling child method')  
class Student:  
    "Test Student Class"  
  
c = Child()  
print(isinstance(c,Parent))  
print(isinstance(c,Child))  
print(isinstance(c,Student))  
  
True  
True  
False
```

## Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes –

Sr.No.	Method, Description & Sample Call
1	<b><u>__init__ ( self [,args...] )</u></b> Constructor (with any optional arguments) Sample Call : <i>obj = className(args)</i>
2	<b><u>__del__( self )</u></b> Destructor, deletes an object Sample Call : <i>del obj</i>

# Conceptualizing Python in Google COLAB

3	<b><code>__repr__( self )</code></b> Evaluable string representation Sample Call : <code>repr(obj)</code>
4	<b><code>__str__( self )</code></b> Printable string representation Sample Call : <code>str(obj)</code>
5	<b><code>__cmp__( self, x )</code></b> Object comparison Sample Call : <code>cmp(obj, x)</code>

## `__str__` → Printable String Representation

When an instance of an class is passed to `print()` method, the module and class name is displayed along with the hash code of the object as shown in the following figure:

```
▶ class Vector:  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
v = Vector(2,10)  
print(v)  
  
<__main__.Vector object at 0x7fab62177e90>
```

However, the `__str__()` method can be overridden in the class which will be invoked when an object is passed to `print()` method for displaying the object in the required string format. In the following example, the `__str__()` method is overridden in ‘Vector’ class to display the x and y coordinates of the vector in parentheses.

# Conceptualizing Python in Google COLAB

```
class Vector:  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
    def __str__(self):  
        return 'Vector (%d, %d)' % (self.a, self.b)  
  
v = Vector(2,10)  
print(v)
```

Vector (2, 10)

In the following example, the `__str__()` method is overridden in ‘Rational’ class to display the numerator and denominator of the rational no. in required format.

```
class Rational:  
    def __init__(self, nr, dr):  
        self.nr = nr  
        self.dr = dr  
    def __str__(self):  
        return str(self.nr) + "/" + str(self.dr)  
r = Rational(2,5)  
print(r)
```

2/5

In the following example, the `__str__()` method is overridden in ‘Complex’ class to display the real and imaginary parts of rational no. in required format.

```
class Complex:  
    def __init__(self, real, img):  
        self.real = real  
        self.img = img  
    def __str__(self):  
        return str(self.real) + "+" + str(self.img) + "i"  
c = Complex(2,5)  
print(c)
```

2+5i

# Conceptualizing Python in Google COLAB

## Object Comparison

When == operator is used for comparing the two reference variables in Python, the addresses of the two objects are compared and not the content. In the following example, ‘n1’ and ‘n2’ are references of ‘Number’ class. Even if their content is same their comparison fails as shown in the following figure:

```
▶ class Number:  
    def __init__(self, val):  
        self.val = val  
  
    def __str__(self):  
        return str(self.val)  
n1 = Number(2)  
n2 = Number(2)  
  
if (n1==n2):  
    print("Equal")  
else:  
    print("Not Equal")  
  
□ Not Equal
```

For comparing the contents of two objects, \_\_eq\_\_() method must be overridden in the class. The following program presents the re-written version of above program which overrides \_\_eq\_\_() method for comparing the content of two ‘Number’ class objects. When \_\_eq\_\_() method is overridden in the class, == operator is mapped to the method for object comparison.

```
▶ class Number:  
    def __init__(self, val):  
        self.val = val  
  
    def __str__(self):  
        return str(self.val)  
  
    def __eq__(self, x):  
        return self.val==x.val  
  
n1 = Number(2)  
n2 = Number(2)  
  
if (n1==n2):  
    print("Equal")  
else:  
    print("Not Equal")  
  
□ Equal
```

# Conceptualizing Python in Google COLAB

---

In the above program, the statement

***n1==n2*** is translated to the statement

***n1.\_\_eq\_\_(n2)***

## Operator Overloading

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. The same built-in operator or function shows different behavior for objects of different classes, this is referred to as *Operator Overloading*.

Python magic methods or special functions for operator overloading

## How to overload the operators in Python?

We can overload all existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method `__add__` is automatically invoked in which the operation for + operator is defined.

The following table depicts the operators that can be overloaded along with magic methods.

# Conceptualizing Python in Google COLAB

OperatorMagic Method	
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>
**	<code>__pow__(self, other)</code>
>>	<code>__rshift__(self, other)</code>
<<	<code>__lshift__(self, other)</code>

&	<code>__and__(self, other)</code>
	<code>__or__(self, other)</code>
^	<code>__xor__(self, other)</code>

Comparison Operators :	
OperatorMagic Method	
<	<code>__LT__(SELF, OTHER)</code>
>	<code>__GT__(SELF, OTHER)</code>
<=	<code>__LE__(SELF, OTHER)</code>
>=	<code>__GE__(SELF, OTHER)</code>
==	<code>__EQ__(SELF, OTHER)</code>
!=	<code>__NE__(SELF, OTHER)</code>

OperatorMagic Method	
<code>-=</code>	<code>__ISUB__(SELF, OTHER)</code>
<code>+=</code>	<code>__IADD__(SELF, OTHER)</code>
<code>*=</code>	<code>__IMUL__(SELF, OTHER)</code>
<code>/=</code>	<code>__IDIV__(SELF, OTHER)</code>
<code>//=</code>	<code>__IFLOORDIV__(SELF, OTHER)</code>
<code>%=</code>	<code>__IMOD__(SELF, OTHER)</code>

# Conceptualizing Python in Google COLAB

<code>**=</code>	<code>_IPOW_(SELF, OTHER)</code>	OperatorMagic Method
<code>&gt;&gt;=</code>	<code>_IRSHIFT_(SELF, OTHER)</code>	
<code>&lt;&lt;=</code>	<code>_ILSHIFT_(SELF, OTHER)</code>	
<code>&amp;=</code>	<code>_IAND_(SELF, OTHER)</code>	
<code> =</code>	<code>_IOR_(SELF, OTHER)</code>	
<code>^=</code>	<code>_IXOR_(SELF, OTHER)</code>	
<code>-</code>	<code>_NEG_(SELF, OTHER)</code>	
<code>+</code>	<code>_POS_(SELF, OTHER)</code>	
<code>~</code>	<code>_INVERT_(SELF, OTHER)</code>	
<code>&lt;</code>		

## Overloading binary + operator in Vector Class

When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method `__add__` is automatically invoked in which the operation for + operator is defined. There by changing this magic method's code, we can give extra meaning to the + operator. In the following example, the `__add__()` method is overloaded in 'Vector' class for defining addition operation on two vector class objects.

# Conceptualizing Python in Google COLAB

```
▶ class Vector:  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b  
  
    def __str__(self):  
        return 'Vector (%d, %d)' % (self.a, self.b)  
  
    def __add__(self, other):  
        return Vector(self.a + other.a, self.b + other.b)  
  
v1 = Vector(2,10)  
v2 = Vector(5,-2)  
print(v1 + v2)  
  
↳ Vector (7, 8)
```

## Overloading binary + operator in Python :

In the following example, the binary operator ‘+’ is overloaded to define addition operation on two objects of ‘*Test*’ class which is tested for addition of two numbers and strings.

```
▶ class Test:  
    def __init__(self,x):  
        self.x=x  
    def __add__(self,obj):  
        return self.x+obj.x  
  
    obj1=Test(10)  
    obj2=Test(20)  
    print(obj1+obj2)  
  
    obj1=Test("Python")  
    obj2=Test(" Programming")  
    print(obj1+obj2)  
  
30  
Python Programming
```

# Conceptualizing Python in Google COLAB

In the following program, binary + operator is overloaded in ‘**Complex**’ class to define addition operation on two objects of ‘**Complex**’ class.

```
class Complex:  
    def __init__(self,real,img):  
        self.real=real  
        self.img=img  
    def __str__(self):  
        return "("+str(self.real)+", "+str(self.img)+")"  
    def __add__(self,obj):  
        return self.real+obj.real,self.img+obj.img  
  
obj1=Complex(10,20)  
obj2=Complex(30,40)  
print("Complex No.1 : ",obj1)  
print("Complex No.2 : ",obj2)  
print("Sum : ",obj1+obj2)
```

Complex No.1 : (10, 20)  
Complex No.2 : (30, 40)  
Sum : (40, 60)

## Python program to overload a comparison operators

The following program demonstrates the overloading of comparison operator in class ‘A’.

```
class A:  
    def __init__(self, a):  
        self.a = a  
    def __gt__(self, other):  
        if(self.a>other.a):  
            return True  
        else:  
            return False  
ob1 = A(2)  
ob2 = A(3)  
if(ob1>ob2):  
    print("ob1 is greater than ob2")  
else:  
    print("ob2 is greater than ob1")
```

ob2 is greater than ob1

# Conceptualizing Python in Google COLAB

## Python program to overload equality and less than operators

The following program demonstrates the overloading of equality and ‘less than’ operator in class ‘A’.

```
class A:  
    def __init__(self, a):  
        self.a = a  
    def __lt__(self, other):  
        if(self.a<other.a):  
            return "ob1 is lessthan ob2"  
        else:  
            return "ob2 is less than ob1"  
    def __eq__(self, other):  
        if(self.a == other.a):  
            return "Both are equal"  
        else:  
            return "Not equal"  
  
ob1 = A(2)  
ob2 = A(3)  
print(ob1 < ob2)  
  
ob3 = A(4)  
ob4 = A(4)  
print(ob1 == ob2)
```

```
↳ ob1 is less than ob2  
Not equal
```

## Data Hiding

An object’s attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

In the following example, the attribute ‘secretCount’ is prefixed with two underscores and hence is not visible outside the class. An attempt to access it outside the class generates ‘AttributeError’ as shown in the following program:

# Conceptualizing Python in Google COLAB

```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print(self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print(counter.__secretCount)

1
2
-----
AttributeError                                 Traceback (most recent call last)
<ipython-input-25-57ff337a14cf> in <module>()
      9 counter.count()
     10 counter.count()
--> 11 print(counter.__secretCount)

AttributeError: 'JustCounter' object has no attribute '__secretCount'

SEARCH STACK OVERFLOW
```

However, the variable ‘secretCount’ can be accessed outside the class using `_JustCounter` as demonstrated in the following program:

```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print(self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print(counter._JustCounter__secretCount)

1
2
2
```

# Conceptualizing Python in Google COLAB

---

## Polymorphism in Python

Polymorphic functions in an object oriented programming language enable same method call responding differently based on the context and how the reference is initialized. The same method call attains different forms.

The basic form of polymorphism is achieved through function overloading. In Python function overloading can be simulated through default parameters. In the following example, the add() method accepts three parameters. The first two parameters are the required parameters and the last parameter is a default parameter. The add() function can thus be invoked with two and three parameters as demonstrated in the following program:

```
def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))

5
9
```

## Example of inbuilt polymorphic functions :

In the following example, the built-in function len() is used for finding both the length of a string and a length of a list. Hence, len() is a polymorphic function.

```
# len() function is used for finding the length of a string
print(len("Python Programming"))

# len() function is used for finding the length of a list
print(len([1,2,3,4,5]))
```

18  
5

# Conceptualizing Python in Google COLAB

## Polymorphism with Class Methods: Adhoc Polymorphism

Class methods can be employed for implementing polymorphism in Python. Depending on the type of the object variable appropriate version of the method is invoked at runtime.

In the following example, each of the classes, ‘**India**’ and ‘**USA**’ have three member functions, *capital()*, *language()* and *type()*. Both the classes are instantiated and the for loop iterates through the tuple containing these instances. Depending on the type of object on which the method is invoked, the appropriate version of the method is invoked as demonstrated in the following program:

```
▶ class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

On execution of the above program, the following output is generated:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

# Conceptualizing Python in Google COLAB

---

## Polymorphism with Inheritance: Classical Polymorphism

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as ***Method Overriding***.

In the following program, Bird is a base class for '**sparrow**' and '**ostrich**' classes which override the '**flight**' method. When the object variable is initialized with instances of 'sparrow' and 'ostrich' classes the overridden methods are invoked as demonstrated in the following program:

```
▶ class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

On execution of the above program, the following output is generated:

# Conceptualizing Python in Google COLAB

```
There are many types of birds.  
Most of the birds can fly but some cannot.  
There are many types of birds.  
Sparrows can fly.  
There are many types of birds.  
Ostriches cannot fly.
```

## Polymorphism with a Function and objects:

It is also possible to create a function that can take any object, allowing for polymorphism.

In the following program, *func()* is a function which accepts a single parameter, a valid instance of a class implementing *capital()*, *language()* and *type()* methods. Depending on the type of object passed to func() method, the *capital()*, *language()* and *type()* methods of the appropriate class are invoked as demonstrated in the following program:

```
class India():  
    def capital(self):  
        print("New Delhi is the capital of India.")  
  
    def language(self):  
        print("Hindi is the most widely spoken language of India.")  
  
    def type(self):  
        print("India is a developing country.")  
  
class USA():  
    def capital(self):  
        print("Washington, D.C. is the capital of USA.")  
  
    def language(self):  
        print("English is the primary language of USA.")  
  
    def type(self):  
        print("USA is a developed country.")  
  
def func(obj):  
    obj.capital()  
    obj.language()  
    obj.type()  
  
obj_ind = India()  
obj_usa = USA()  
  
func(obj_ind)  
func(obj_usa)
```

# Conceptualizing Python in Google COLAB

---

On execution of the above program, the following output is generated:

```
New Delhi is the capital of India.  
Hindi is the most widely spoken language of India.  
India is a developing country.  
Washington, D.C. is the capital of USA.  
English is the primary language of USA.  
USA is a developed country.
```

## Using super

Python super() function provides us the facility to refer to the parent class explicitly. It is basically useful where we have to call superclass functions. It returns the proxy object that allows us to refer parent class by ‘super’.

### Example:

In the following example, class ‘A’ has a single data member ‘x’ and a **display()** method to display the value of ‘x’. ‘B’ is a derived class of ‘A’ with the new data member ‘y’. Hence, the class ‘B’ has access to two data members:

- ‘x’ inherited from class ‘A’
- ‘y’ newly added by class ‘B’

The class ‘B’ has **display()** method to display the values of ‘x’ and ‘y’ which uses ‘super’ keyword for invoking the **display()** method of parent class ‘A’ and then displays the value of ‘y’.

‘C’ is a derived class of ‘B’ with the new data member ‘z’. Hence, the class ‘C’ has access to three data members:

- ‘x’ inherited from class ‘A’
- ‘y’ inherited from class ‘A’
- ‘z’ newly added by class ‘C’

The class ‘C’ has **display()** method to display the values of ‘x’, ‘y’ and ‘z’ which uses ‘super’ keyword for invoking the **display()** method of parent class ‘B’ for displaying the values of ‘x’ and ‘y’ and then displays the value of ‘z’.

# Conceptualizing Python in Google COLAB

Hence each class is responsible for performing the responsibility assigned to it. The complete source code of the program is shown below:

```
class A:  
    def __init__(self,x):  
        self.x=x  
    def display(self):  
        print("x :",self.x)  
  
class B(A):  
    def __init__(self,x,y):  
        A.__init__(self,x)  
        self.y=y  
    def display(self):  
        super().display()  
        print("y :",self.y)  
  
class C(B):  
    def __init__(self,x,y,z):  
        B.__init__(self,x,y)  
        self.z=z  
    def display(self):  
        super().display()  
        print("z :",self.z)  
  
obj1=C(10,20,30)  
obj1.display()
```

```
↳ x : 10  
    y : 20  
    z : 30
```

## Implementation of Complex Class

In the following program, the complex class is implemented with the following functionality:

- Overloading == operator (`__eq__`)
- Overloading + operator (`__add__`)
- Overloading - operator (`__sub__`)
- Stringifying object (`__str__`)

# Conceptualizing Python in Google COLAB

```
class Complex:  
    def __init__(self, real, img):  
        self.real = real  
        self.img = img  
    def __str__(self):  
        return str(self.real) + "+" + str(self.img) + "i"  
  
    def __add__(self,c):  
        return str(self.real+c.real) + "+" + str(self.img+c.img) + "i"  
  
    def __sub__(self,c):  
        return str(self.real-c.real) + "+" + str(self.img-c.img) + "i"  
  
    def __eq__(self,c):  
        return self.real == c.real and self.img == c.img  
  
c1 = Complex(2,5)  
c2 = Complex(3,6)  
c3 = c1 + c2  
c4 = c1 - c2  
  
c5 = Complex(3,6)  
print(c1)  
print(c2)  
print(c3)  
print(c4)  
if (c2==c5):  
    print("c2 ad c5 are equal")  
else:  
    print("c2 ad c5 are not equal")
```

```
if (c1==c2):  
    print("c1 ad c2 are equal")  
else:  
    print("c11 ad c2 are not equal")  
  
2+5i  
3+6i  
5+11i  
-1+-1i  
c2 ad c5 are equal  
c11 ad c2 are not equal
```

# Conceptualizing Python in Google COLAB

---

## Case Study:

### Comparison of Rational No.s –

For comparing the two rational no.s the following cases need to be considered. The rational no.s may have different signs in their numerator and denominator. So both the rational no.s to be compared need to be standardized. The denominators of two rational no.s to be compared may not be same, so they need to be normalized. The different cases to be dealt with are depicted below:

Case 1: If signs are not equal, then +ve is greater than -ve

Case 2: Signs are equal and both +ve, denominators are equal.

Case 3: Signs are equal and both +ve, denominators are not equal.

Case 4: Signs are equal and both -ve, denominators are equal.

Case 5: Signs are equal and both -ve, denominators are not equal.

Case 6: If signs are not equal and denominators are equal

Case 7: If signs are not equal and denominators are not equal

Test Case 1:

- i.       $\frac{2}{3}$  and  $-\frac{4}{5}$
- ii.      $-\frac{2}{3}$  and  $\frac{4}{5}$

Test Case 2:

$\frac{2}{3}$  and  $\frac{4}{3}$

Test Case 3:

$\frac{2}{3}$  and  $\frac{4}{5}$

Test Case 4:

$-\frac{2}{3}$  and  $-\frac{4}{3}$

Test Case 5:

$-\frac{2}{3}$  and  $-\frac{4}{5}$

Test Case 6:

$\frac{2}{3}$  and  $-\frac{4}{3}$

Test Case 7:

# Conceptualizing Python in Google COLAB

2/3 and -4/5

## Standardizing Rational No.s

The following code depicts the standardization procedure before comparing the two rational nos. If the denominator of a rational no is negative, then the signs of numerator and denominator are reversed.

```
class Rational:
    def __init__(self, nr, dr):
        self.nr = nr
        self.dr = dr
    def __str__(self):
        return str(self.nr) + "/" + str(self.dr)
    def __gt__(self, r):
        if (self.dr < 0):
            self.nr=-self.nr
            self.dr=-self.dr
        if (r.dr < 0):
            r.nr=-r.nr
            r.dr=-r.dr
        print("After Standardization : ",self)
        print("After Standardization : ",r)

r1 = Rational(2,-3)
r2 = Rational(-4,-5)
r1>r2
```

```
After Standardization : -2/3
After Standardization : 4/5
```

## Normalizing Rational No.s

The following code depicts the normalization of two rational no.s having different denominators. If nr1 and dr1 are numerator and denominator of first rational no. and nr2 and dr2 are numerator and denominator of second rational no. and if dr1 is not equal to dr2, then both numerator and denominator of first rational no. are multiplied by dr2 and both numerator and denominator of second rational no. are multiplied by dr1.

Hence the first rational no. is

$$\frac{nr1 * dr2}{dr1 * dr2}$$

# Conceptualizing Python in Google COLAB

and the second rational no is

$$\frac{nr2 * dr1}{dr2 * dr1}$$

Now, the denominators of both the rational no.s are equal and they can now be compared directly by comparing their respective numerators.

The complete source code for the implementation of the above case study is shown below along with the execution of individual cases.

```
class Rational:
    def __init__(self, nr, dr):
        self.nr = nr
        self.dr = dr
    def __str__(self):
        return str(self.nr) + "/" + str(self.dr)
    def __gt__(self,r):
        if (self.dr < 0):
            self.nr=-self.nr
            self.dr=-self.dr
        if (r.dr < 0):
            r.nr=-r.nr
            r.dr=-r.dr
        print("After Standardization : ",self)
        print("After Standardization : ",r)
        if (self.dr==r.dr):
            if(self.nr > r.nr):
                return True
                #print(self," is greater than ",r)
            else:
                return False
                #print(r," is greater than ",self)
        else:
            dr=self.dr
            self.nr=self.nr*r.dr
            self.dr=self.dr*r.dr
            r.nr=r.nr*dr
            r.dr=r.dr*dr
            print("After Normalization : ",self)
            print("After Normalization : ",r)
            if(self.nr > r.nr):
                return True
```

# Conceptualizing Python in Google COLAB

```
#print(self," is greater than ",r)
else:
    return False
#print(r," is greater than ",self)

r1 = Rational(2,5)
r2 = Rational(-4,5)
if (r1>r2):
    print(r1, " is greater than ",r2)
else:
    print(r2, " is greater than ",r1)
```

```
After Standardization :  2/5
After Standardization : -4/5
2/5  is greater than -4/5
```

Test Case 1:

```
↳ After Standardization :  2/3
After Standardization : -4/5
After Normalization :  10/15
After Normalization : -12/15
10/15  is greater than -12/15
```

```
After Standardization : -2/3
After Standardization :  4/5
After Normalization : -10/15
After Normalization :  12/15
12/15  is greater than -10/15
```

Test Case 2:

```
After Standardization :  2/3
After Standardization :  4/3
4/3  is greater than  2/3
```

# Conceptualizing Python in Google COLAB

---

Test Case 3:

```
After Standardization : 2/3
After Standardization : 4/5
After Normalization : 10/15
After Normalization : 12/15
12/15  is greater than 10/15
```

Test Case 4:

```
After Standardization : -2/3
After Standardization : -4/3
-2/3  is greater than -4/3
```

Test Case 5:

```
After Standardization : -2/3
After Standardization : -4/5
After Normalization : -10/15
After Normalization : -12/15
-10/15  is greater than -12/15
```

Test Case 6:

```
After Standardization : 2/3
After Standardization : -4/3
2/3  is greater than -4/3
```

Test Case 7:

```
After Standardization : 2/3
After Standardization : -4/5
After Normalization : 10/15
After Normalization : -12/15
10/15  is greater than -12/15
```

# Conceptualizing Python in Google COLAB

---

## References:

1. <https://docs.python.org/3/tutorial/>
2. <https://www.tutorialspoint.com/python/index.htm>
3. <https://www.w3schools.com/python/>
4. <https://www.javatpoint.com/python-tutorial>
5. <https://www.programiz.com/python-programming>
6. <https://www.learnpython.org/>
7. <https://www.geeksforgeeks.org/python-programming-language/learn-python-tutorial/>
8. <https://realpython.com/tutorials/python/>
9. [https://colab.research.google.com/?utm\\_source=scs-index](https://colab.research.google.com/?utm_source=scs-index)

## Appendix A

### Case Study on Object Oriented Programming

#### Level – Intermediate

#### Implementation of Custom Signed Number Class

##### Problem Definition:

Define a class with the name '**Number**' with the following data members –

- value
- sign

Overload constructors for creating both signed and unsigned objects of 'Number' class, as shown below –

- 0 (No Sign)
- +10
- -20

Define the following operations on the objects of '**Number**' class –

- `__str__()` method for displaying value along with proper sign
- increment
- decrement
- addition
- subtraction
- multiplication
- division
- max
- min

# Conceptualizing Python in Google COLAB

---

Addition operation on objects of ‘Number’ class

**Case 1 –**

If ‘sign’ attribute of both ‘Number’ objects is +ve, then result is addition of two values and sign is +ve.

**Case 2 –**

If ‘sign’ attribute of both ‘Number’ objects is -ve, then result is addition of two values and sign is -ve.

**Case 3 –**

If ‘sign’ attribute of ‘Number’ objects are different, then result is absolute difference of two values and sign equal to the ‘sign’ attribute of ‘Number’ object whose ‘value’ attribute is greater. If the value is zero, then sign is blank.

Increment operation on objects of ‘Number’ class

**Case 1 –**

If ‘sign’ attribute of ‘Number’ object is +ve, then result is addition of one to value attribute and sign is +ve.

**Case 2 –**

If ‘sign’ attribute of ‘Number’ object is -ve, then result is subtraction of one from value attribute and sign is -ve if ‘value’ attribute is non-zero else sign is blank .

**Case 3 –**

If value attribute of a ‘Number’ object is zero, then result is a Number object with value attribute equal to 1 and sign attribute equal to +ve.

# Conceptualizing Python in Google COLAB

---

Test Cases –

+1	+2
-1	0
-2	-1
0	+1

Decrement operation on objects of ‘Number’ class

Case 1 –

If ‘sign’ attribute of ‘Number’ object is +ve, then result is subtraction of one from value attribute and sign is +ve.

Case 2 –

If ‘sign’ attribute of ‘Number’ object is -ve, then result is subtraction of one from value attribute and sign is -ve if ‘value’ attribute is non-zero else sign is blank .

Maximum operation on objects of ‘Number’ class

Test Cases –

+1	+2
-1	0
+1	0
-2	-1
+2	-1
-2	+1
0	+1
0	-1

Note – Return maximum ‘Number’ class object

Solution:

Solution – Number Class

# Conceptualizing Python in Google COLAB

---

```
class Number:
```

```
    def __init__(self,value=0,sign=""):  
        self.value=value  
        self.sign=sign  
    def __str__(self):  
        return self.sign+str(self.value);
```

```
obj=Number()
```

```
print(obj)
```

```
obj1=Number(10,"+")
```

```
print(obj1)
```

```
obj2=Number(20,"-")
```

```
print(obj2)
```



The screenshot shows a Google Colab cell with the following code:

```
class Number:  
    def __init__(self,value=0,sign=""):  
        self.value=value  
        self.sign=sign  
    def __str__(self):  
        return self.sign+str(self.value);  
  
obj=Number()  
print(obj)  
obj1=Number(10,"+")  
print(obj1)  
obj2=Number(20,"-")  
print(obj2)
```

The output of the code is:

```
0  
+10  
-20
```

# Conceptualizing Python in Google COLAB

---

## Implementation of Addition Operation

class Number:

```
def __init__(self,value=0,sign=""):  
    self.value=value  
    self.sign=sign  
  
def __str__(self):  
    return self.sign+str(self.value);  
  
  
def __add__(self,other):  
    if (self.sign=="+" and other.sign=="+"):  
        return Number(self.value+other.value,"+")  
  
    elif(self.sign=="-" and other.sign=="-"):  
        return Number(self.value+other.value,"-")  
  
    else:  
        value=abs(self.value-other.value)  
        if (self.value>other.value):  
            sign=self.sign  
        else:  
            sign=other.sign  
  
        return Number(value,sign)  
  
obj1=Number(10,"+")  
  
obj2=Number(20,"+")  
  
obj3=obj1+obj2  
  
print(obj3)
```

# Conceptualizing Python in Google COLAB

---

```
obj4=Number(10,"-")
obj5=Number(20,"-")
obj6=obj4+obj5
print(obj6)

obj7=Number(10,"-")
obj8=Number(20,"+")
obj9=obj7+obj8
print(obj9)

obj10=Number(10,"+")
obj11=Number(20,"-")
obj12=obj10+obj11
print(obj12)
```

The screenshot shows a Google Colab cell with a play button icon. The code defines a `Number` class with methods for initialization, string representation, and addition. The addition method handles both positive and negative signs, returning the correct value and sign.

```
class Number:
    def __init__(self,value=0,sign=""):
        self.value=value
        self.sign=sign
    def __str__(self):
        return self.sign+str(self.value);

    def __add__(self,other):
        if (self.sign=="+" and other.sign=="+"):
            return Number(self.value+other.value,"+")
        elif(self.sign=="-" and other.sign=="-"):
            return Number(self.value+other.value,"-")
        else:
            value=abs(self.value-other.value)
            if (self.value>other.value):
                sign=self.sign
            else:
                sign=other.sign
            return Number(value,sign)
```

# Conceptualizing Python in Google COLAB

```
obj1=Number(10,"+")
obj2=Number(20,"+")
obj3=obj1+obj2
print(obj3)

obj4=Number(10,"-")
obj5=Number(20,"-")
obj6=obj4+obj5
print(obj6)

obj7=Number(10,"-")
obj8=Number(20,"+")
obj9=obj7+obj8
print(obj9)

obj10=Number(10,"+")
obj11=Number(20,"-")
obj12=obj10+obj11
print(obj12)
```

```
↳ +30
-30
+10
-10
```

## Implementation of Increment Operation

class Number:

```
def __init__(self,value=0,sign ""):
    self.value=value
    self.sign=sign
def __str__(self):
    return self.sign+str(self.value)
```

# Conceptualizing Python in Google COLAB

---

```
def increment(self):
    if(self.sign=="+"):
        self.value=self.value+1
    elif (self.sign=="-" and self.value==1):
        self.sign=""
        self.value=0
    elif (self.sign=="-"):
        self.sign="-"
        self.value=self.value-1
    elif (self.value==0):
        self.sign="+"
        self.value=1
obj=Number(10,"+")
obj.increment()
print(obj)

obj1=Number(1,"-")
obj1.increment()
print(obj1)

obj2=Number(2,"-")
obj2.increment()
print(obj2)

obj3=Number(0,"")
obj3.increment()
print(obj3)
```

# Conceptualizing Python in Google COLAB

```
class Number:  
    def __init__(self,value=0,sign=""):  
        self.value=value  
        self.sign=sign  
    def __str__(self):  
        return self.sign+str(self.value)  
  
    def increment(self):  
        if(self.sign=="+"):  
            self.value=self.value+1  
        elif (self.sign=="-" and self.value==1):  
            self.sign=""  
            self.value=0  
        elif (self.sign=="-"):  
            self.sign="-"  
            self.value=self.value-1  
        elif (self.value==0):  
            self.sign="+"  
            self.value=1
```

```
obj=Number(10,"+")
obj.increment()
print(obj)

obj1=Number(1,"-")
obj1.increment()
print(obj1)

obj2=Number(2,"-")
obj2.increment()
print(obj2)

obj3=Number(0,"")
obj3.increment()
print(obj3)
```

```
↳ +11
  0
  -1
  +1
```

# Conceptualizing Python in Google COLAB

---

## Implementation of Max Operation

class Number:

```
def __init__(self,value=0,sign=""):
```

```
    self.value=value
```

```
    self.sign=sign
```

```
def __str__(self):
```

```
    return self.sign+str(self.value)
```

```
def max(self,other):
```

```
    if(self.value==0 and other.value==0):
```

```
        return self
```

```
    elif(self.sign=="+" and other.sign=="+"):
```

```
        if(self.value > other.value):
```

```
            value=self.value
```

```
        else:
```

```
            value=other.value
```

```
        sign="+"
```

```
    return Number(value,sign)
```

```
elif(self.value==0):
```

```
    if(other.value>0 and other.sign=="+"):
```

```
        return other
```

```
    if(other.value>0 and other.sign=="-"):
```

```
        return self
```

```
elif(other.value==0):
```

# Conceptualizing Python in Google COLAB

---

```
if(self.value>0 and self.sign=="+"):  
    return self  
  
if(self.value>0 and self.sign=="-"):  
    return other  
  
elif(self.sign=="-" and other.sign=="-"):  
    if(self.value < other.value):  
        return self  
  
    else:  
        return other  
  
elif (self.sign != other.sign):  
    if(self.sign == "+"):  
        return self  
  
    else:  
        return other  
  
obj1=Number(10,"+")  
  
obj2=Number(20,"+")  
  
obj3=obj1.max(obj2)  
  
print(obj3)  
  
  
obj4=Number(0,"")  
  
obj5=Number(10,"+")  
  
obj6=obj4.max(obj5)  
  
print(obj6)  
  
obj7=Number(0,"")  
  
obj8=Number(10,"-")
```

# Conceptualizing Python in Google COLAB

---

```
obj9=obj7.max(obj8)
```

```
print(obj9)
```

```
obj10=Number(10,"+")
```

```
obj11=Number(0,"")
```

```
obj12=obj10.max(obj11)
```

```
print(obj12)
```

```
obj13=Number(10,"-")
```

```
obj14=Number(0,"")
```

```
obj15=obj13.max(obj14)
```

```
print(obj15)
```

```
obj16=Number(0,"")
```

```
obj17=Number(0,"")
```

```
obj18=obj16.max(obj17)
```

```
print(obj18)
```

```
obj19=Number(10,"-")
```

```
obj20=Number(20,"-")
```

```
obj21=obj19.max(obj20)
```

```
print(obj21)
```

```
obj22=Number(10,"+")
```

```
obj23=Number(20,"-")
```

```
obj24=obj22.max(obj23)
```

```
print(obj24)
```

# Conceptualizing Python in Google COLAB

---

```
obj25=Number(10,"-")
obj26=Number(20,"+")
obj27=obj25.max(obj26)
print(obj27)
```

```
class Number:
    def __init__(self,value=0,sign ""):
        self.value=value
        self.sign=sign
    def __str__(self):
        return self.sign+str(self.value)

    def max(self,other):
        if(self.value==0 and other.value==0):
            return self
        elif(self.sign=="+" and other.sign=="+"):
            if(self.value > other.value):
                value=self.value
            else:
                value=other.value
            sign="+"
            return Number(value,sign)
        elif(self.value==0):
            if(other.value>0 and other.sign=="+"):
                return other
            if(other.value>0 and other.sign=="-"):
                return self
        elif(other.value==0):
            if(self.value>0 and self.sign=="+"):
                return self
            if(self.value>0 and self.sign=="-"):
                return other
        elif(self.sign=="-" and other.sign=="-"):
            if(self.value < other.value):
                return self
            else:
                return other
        elif (self.sign != other.sign):
```

# Conceptualizing Python in Google COLAB

---

```
if(self.sign == "+"):
    return self
else:
    return other
```

```
obj1=Number(10,"+")
obj2=Number(20,"+")
obj3=obj1.max(obj2)
print(obj3)

obj4=Number(0,"")
obj5=Number(10,"+")
obj6=obj4.max(obj5)
print(obj6)

obj7=Number(0,"")
obj8=Number(10,"-")
obj9=obj7.max(obj8)
print(obj9)

obj10=Number(10,"+")
obj11=Number(0,"")
obj12=obj10.max(obj11)
print(obj12)

obj13=Number(10,"-")
obj14=Number(0,"")
obj15=obj13.max(obj14)
print(obj15)

obj16=Number(0,"")
obj17=Number(0,"")
obj18=obj16.max(obj17)
print(obj18)
```

# Conceptualizing Python in Google COLAB

```
obj19=Number(10,"-")
obj20=Number(20,"-")
obj21=obj19.max(obj20)
print(obj21)

obj22=Number(10,"+")
obj23=Number(20,"-")
obj24=obj22.max(obj23)
print(obj24)

obj25=Number(10,"-")
obj26=Number(20,"+")
obj27=obj25.max(obj26)
print(obj27)
```

```
[+] +20
    +10
    0
    +10
    0
    0
    -10
    +10
    +20
```

## Thought for Case Study 2 -

RationalNumber Class – Modeling Relationship Between Number and RationalNumber Classes

What is relationship between Number and RationalNumber class?

Inheritance

Delegation?

Delegation

Numerator and denominator are ‘Number’ class objects.

Delegate operations to objects of ‘Number’ class.

Number nr

# Conceptualizing Python in Google COLAB

---

Number dr

2/3 4/5

r1.nr/ r1.dr

r2.nr/r2.dr

Second Case Study –

class Number:

```
def __init__(self,value=0,sign=""):  
    self.value=value  
    self.sign=sign  
def display(self):  
    print(self.sign+str(self.value),end="")
```

class Rational:

```
def __init__(self, nr, nr_sign, dr, dr_sign):  
    self.nr=Number(nr, nr_sign)  
    self.dr=Number(dr, dr_sign)  
def display(self):  
    self.nr.display()  
    print("/",end="")  
    self.dr.display()
```

r1=Rational(10,"+",20,"-")

r1.display()

# Conceptualizing Python in Google COLAB

```
class Number:  
    def __init__(self,value=0,sign=""):  
        self.value=value  
        self.sign=sign  
    def display(self):  
        print(self.sign+str(self.value),end="")  
  
class Rational:  
    def __init__(self, nr, nr_sign, dr, dr_sign):  
        self.nr=Number(nr, nr_sign)  
        self.dr=Number(dr, dr_sign)  
    def display(self):  
        self.nr.display()  
        print("/",end="")  
        self.dr.display()  
  
r1=Rational(10,"+",20,"-")  
r1.display()
```

+10/-20

class Number:

```
def __init__(self,value=0,sign=""):  
    self.value=value  
    self.sign=sign  
  
def __str__(self):  
    return self.sign+str(self.value)
```

class Rational:

```
def __init__(self, nr, nr_sign, dr, dr_sign):  
    self.nr=Number(nr, nr_sign)  
    self.dr=Number(dr, dr_sign)  
  
def __str__(self):
```

# Conceptualizing Python in Google COLAB

---

```
print(self.nr,end="")
print("/",end="")
print(self.dr)
return ""
```

```
r1=Rational(10,"+",20,"-")
```

```
print(r1)
```

```
class Number:
    def __init__(self,value=0,sign ""):
        self.value=value
        self.sign=sign
    def __str__(self):
        return self.sign+str(self.value)

class Rational:
    def __init__(self,nr(nr_sign),dr,dr_sign):
        self.nr=Number(nr,nr_sign)
        self.dr=Number(dr,dr_sign)
    def __str__(self):
        print(self.nr,end="")
        print("/",end="")
        print(self.dr)
        return ""

r1=Rational(10,"+",20,"-")
print(r1)
```

```
+10/-20
```

# About the Book

This book can serve as textbook for post graduates and reference for any computer graduate. It will also provide easy reference for Computer Professionals who wants to begin their career in Machine Learning using Python. This book is precisely organized into twelve chapters. Each chapter has been carefully developed with the help of several implemented concepts. Dedicated efforts have been put in to ensure that every concept of Python discussed in this book is explained with help of relevant commands and screenshots of the outputs have been included. Chapter 1 focuses on development environment offered by Google COLAB. Chapters 2 through 4 cover the Python language fundamentals focusing on control and iterative statements, operators along with their applications in basic programs. Python employs blended programming paradigm in which it is procedural, object-oriented and functional. The best part of all programming languages reside in a single platform. Chapter 5 focuses on functions in Python with a special emphasis on Lambda functions. Advanced Python programming concepts such as iterators, closures, decorators, generators are covered at depth in Chapter 6 and 7. A good and in-depth knowledge of exception handling enables in writing a reliable and robust code. To cater to this need Chapter 8 unleashes the salient features of exception handling in Python. Data persistence through file handling is covered in Chapter 9. Due to the wide application of Regular expressions in pattern matching, Chapter 10 is fully devoted to understanding of regular expression in Python. Different types of common errors that might creep in during the execution of a Python program are summarized in Chapter 11. Final Chapter 12 is devoted to implementation of object oriented concepts in Python. The case study based on object oriented concept is discussed at depth and implemented in Appendix A.

