Written by
Bemnet Girma

Fun-damentals of

PYTHON

PROGRAMMING

Gateway for the Joy of Programming

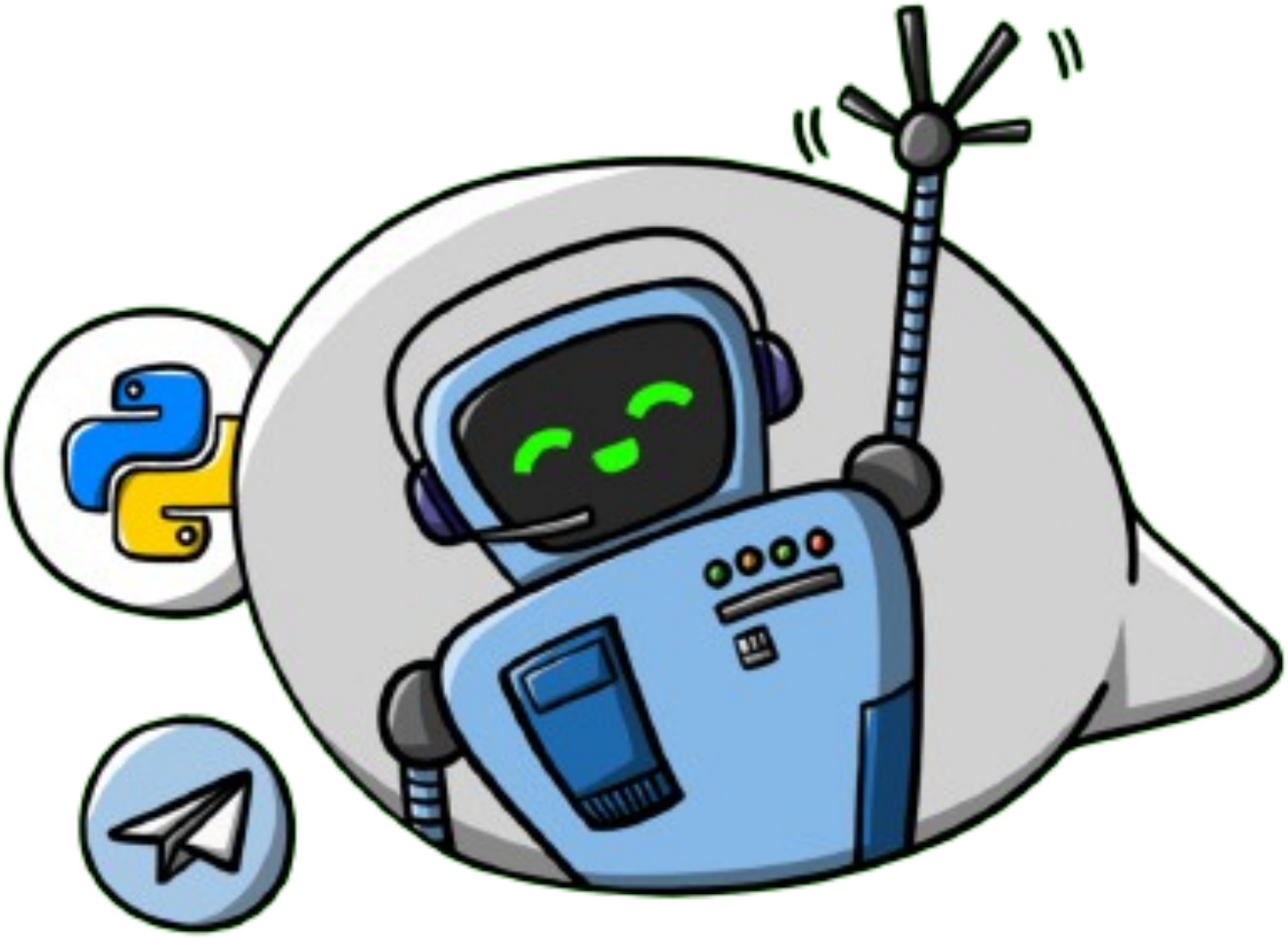
Digitally Published Book

No content here - this page is intentionally blank.

Written by
Bemnet Girma

Fun-damentals of

PYTHON



PROGRAMMING

Gateway for the Joy of Programming

Digitally Published Book

Contacts
bemnet.dev@gmail.com
https://www.linkedin.com/in/bemnetdev/

Publisher's Cataloging-in-Publication Data

Fun-damentals of Python Programming
Gateway for the Joy of Programming
Bemnet Girma Sahilu. — 1st ed.
First Edition, April 2024.

CONTENTS

No content here - this page is intentionally blank.

INTRODUCTION

Everybody should learn to program a computer, because it teaches you how to think.

— STEVE JOBS

Computer programming used to solve problems. However It is still taught as boring theory and most beginners fails to learn then they hate it. Here, I tried to fluid the boring theory. This is my humble attempt to help the world, by pushing you to love programming.

Why Python?

Python is an easy-to-learn programming language that has some really useful features for a beginner. The code is quite easy to read when compared to other programming languages, and you can solve and make almost anything with it.

How to learn to code?

Like anything you try for the first time, it's always best to start with the basics. so begin with the first chapters and resist the urge to skip ahead to the later chapters.

PS. The Expert in anything was once a beginner.

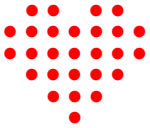
If you jump ahead too quickly, not only will the basic ideas not stick in your head, but you'll also find the content of the later chapters more complicated than it actually is. As you go through this book, try each of the examples, so you can see how they work. You can run the code in this book using free Google Colab links to practice and experiment.

Who should read this book?

This book is for everyone – from complete newbies to experienced programmers looking for a refresher. By the time you reach the end, you'll be writing your own programs, solving puzzles, and get ready to use other python libraries and frameworks. Most importantly, you'll rediscover the joy of learning and the thrill of bringing your ideas to life through code.

Have Fun!

Think programming as a way to create some fun games or playing puzzles with your friends. Learning to program is a wonderful mental exercise and the results can be very rewarding. But most of all, whatever you do, have fun!



Have a nice journey,

Bennet

No content here - this page is intentionally blank.

CHAPTER ONE

Get Started

Objectives

- *What is programming?*
- *What is programming language?*
- *How computers understand a programming language?*
- *How to write and run a code?*

Lesson One : Introduction to Programming

Computers are dumb. Computers are only smart because we program them to be.

What is Programming?

Programming is the process of preparing an instructional program for a device to do some task without making mistakes.

There are two types of languages for communication between a computer and a programmer (human).

1. Machine language: which is a binary code (series of 0s and 1s) and computers understood it and it is difficult for humans to understand.

2. Programming language: which is a source code and humans understood it and computers could not.

What is Programming Language?

A programming language is a formal language, which includes a set of instructions that helps a programmer to communicate or interact with a computer.

Programming Language is classified into two categories, based on the level (Degree) of abstraction and gap between source code and machine code.

1. Source code is any collection of code, written using a human-readable programming language, usually as plain text.

2. Machine code is a computer program written in machine language instructions that can be executed directly by a computer's control unit.

High-level language Vs low-level language

High level programming languages are those which are programmer friendly. Humans can easily understand it which helps in easy maintaining and debugging of the code.

> Examples – Java, C, C++, Python

Low level programming languages are those which are machine friendly. Humans can't understand such languages. In other words, low level languages are not human readable.

> Examples – Assembly language, Machine language.

PS. Computers only understand machine code.

Lesson Two : Compiler and Interpreter

What is Compiler?

A compiler is a computer program that translates source code written in one programming language (the source language) into another language (the target language) to create an executable program.

What is Interpreter?

An interpreter is a computer program that directly executes instructions written in a programming language, without requiring them previously to have been compiled into a machine language program.

Compiler Vs Interpreter

All the programming languages are converted to machine understandable language for execution. The conversion is taken care of by compilers and interpreters.

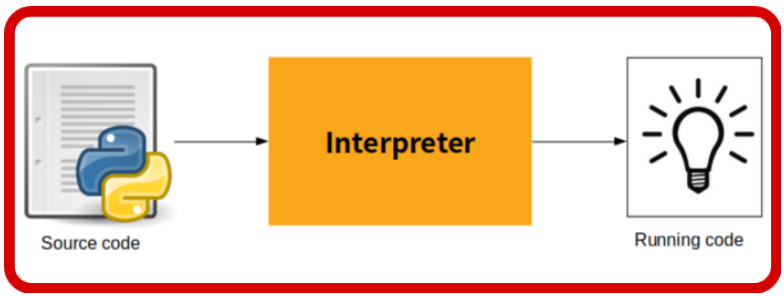
A compiler takes the source code and converts it into machine executable code at once. The processor takes the executable and executes it. The languages which use a compiler are called compiled languages. Compiled languages are faster and more efficient comparatively.

> Examples – Java, C++

An interpreter is one which runs the code line by line and executes instruction by instruction. The languages which use interpreters are called Interpreted languages. Such languages are slower in terms of execution and less efficient.

💡 *Suppose you want to watch an Italian movie, but you don't understand the Italian language. However, you can understand English. You have two options to watch the movie. The first way is to watch an English dubbed version of the movie, if available. The second way is to watch the Italian movie with English subtitles. In the second way, the subtitles are displayed scene by scene separately, whereas in the first way, all the scenes are converted to English. We can compare the first scenario with a compiler and the second one with an interpreter.*

> Examples – PHP, Python, Ruby



Lesson Three : How to write and run a code?

How to write a code?

To write and execute a python program on our computer we need to install two basic Application softwares called Editor (IDE) and compiler on our computer. The editor is used to write our source code and compiler used to compile and execute our source code. There are many editors and compilers available for Python programming. You need to download any one. But here we use Jupyter Notebook as an IDE.

What is IDE?

IDE (Integrated Development Environment) is a place (graphic interface) to write, run, and debug (check errors) code and also convert your source code to machine code.

> Examples: Visual Studio, IDLE, PyCharm, Jupyter Notebook

How to run (execute) a code?

You can't simply type rubbish in an IDE, convert it to machine code and expect the computer to understand it. Each programming language has its own set of rules (Grammar) you must follow within an IDE called Syntax.

What is Syntax?

Syntax is set of rules you must follow to run your program correctly. Syntax for each programming language is unique.

PS. Breaking programming rules will result in an error.

> Example:
> Mens clothes are cheap.
> Men's clothes are cheap

Key Takeaways

- *Programming is the process of preparing an error-free program for a device.*
- *A programming language is a formal language to communicate with a computer through instructions.*
- *Programming languages are translated to machine-readable language by compilers and interpreters.*
- *Compiler is a program that translates code written in one programming language into a language that a computer can understand and execute.*
- *Interpreter is a program that directly executes code written in a programming language without prior translation into a machine-readable form.*
- *To write and execute a python program on our computer we need to install IDE and compiler.*
- *IDE is a graphic interface to write, run, debug and also convert code to machine code.*
- *Syntax is set of rules you must follow to run your program correctly.*

// END OF CHAPTER ONE

Click here for Chapter Two

CHAPTER TWO

Hello Python

Objectives

- *What is Python?*
- *Why we study Python?*
- *What are the key features of Python?*
- *What are applications of Python?*
- *How to install jupyter notebook?*

Lesson One : Introduction to Python

To better communicate with a computer you should have to learn a programming language, Even though you have translators like compiler and interpreter. But which programming language? The answer would be simple, used for different / multiple purposes, i.e. Python.

What is Python?

Python is simple, general purpose, dynamic, object oriented, interpreted as well as compiled, high-level programming language.

In details Python is

- **Simple** – It is easy to learn and code.
- **General purpose** – It can be used to build just about anything.
- **High level language** – It is human readable language which easily understandable and
- **Easy to debug** – It is easy to check and correct your error.
- **Dynamic** – You don't need to write everything to instruct your computer.
- **Object Oriented** – Structure of python program can be grouped based on its behavior.
- **Interpreted as well as compiled** – Python is basically called an interpreted language, but can also be called a compiled interpreted language.

In Python, first source code compiles into a bytecode then bytecode is sent for execution to PVM (Python Virtual Machine). The PVM is an interpreter that runs the bytecode.

Why we study Python?

Because Python is

- One of easiest language
- Fastest growing language
- Vast implementation areas
- Used in many companies

History or Origin of Python

- It was invented in the Netherlands in the early 90's.
- Guido Van Rossum was the creator of this beautiful language.
- Guido released the first version in 1991.
- Python was derived from ABC programming language, a general purpose programming language.
- It is open source software which can be downloaded freely and the code is customizable as well.

Why the name Python?

💡 *There was a TV show by the name Monty Python's Flying Circus which was a very much popular fun show in the 1970's. While creating Python, Guido also used to read this show's published scripts. Guido needed a short and a different name for this language, hence he named it "Python".*

Python Versions

- Python 1.0V introduced in Jan 1994
- Python 2.0V introduced in October 2000
- Python 3.0V introduced in December 2008.
- Current version is 3.12 (Feb 2024)

PS. Python 3 won't provide backward compatibility to Python2 i.e. there is no guarantee that Python2 programs will run in Python3.

Features of Python

- **Simple** - Python syntax is very easy. Developing and understanding python is very easy than others. The below comparison illustrated how simple python language is when compared to other languages.

> **Hello World Program in C language**
>
> ```
> # include <stdio.h>
> int main()
> {
>     printf("Hello World")
>     return 0
> }
> ```
>
> **Hello World Program in Python**
>
> ```
> print("Hello World")
> ```

- **Open Source** - We can download freely and customize the code as well
- **Platform independent** - Python programs are not dependent on any specific operating systems. We can run on all operating systems happily.
- **Portable** - If a program gives the same result on any platform then it is a portable program. Python used to give the same result on any platform.
- **Huge library** - Python has a big library to fulfil the requirements.
- **Database connectivity** - Python provides interfaces to connect with all major databases like, oracle, MySQL.

Lesson Two : Applications of Python

Python is a programming language that does it all, from web applications to video-games, Data Science, Machine Learning, real-time applications to embedded applications, and so much more. In this lesson, we'll take a deeper dive into a broader list of applications of Python out in the wild.

1. Web Development

I hope you are familiar with what web development is. Python is used to develop fully functional website using web development frameworks like Flask and Django. Using Python for web development also offers several other benefits, such as security, easy scalability, and convenience in the development process.

2. Game Development

Just like web development, Python comes equipped with an arsenal of tools and libraries for 2D, 3D and video game development like PyGame.

3. AI and Machine Learning

Artificial Intelligence(AI) and Machine Learning are undoubtedly among the hottest topics of this decade. These are the brains behind the smart tech that we so rely on today to help us make optimized decisions. Python has seen a steep increase in their use for developing AI and ML-powered solutions like YouTube video recommendations. We have TensorFlow, PyTorch, Pandas, Scikit-Learn, NumPy, SciPy, and more libraries.

4. Desktop and Mobile Applications

Python offers plenty of options to desktop and mobile application developers to build super-fast, responsive and fully functional GUI using tools like Tkinter and Kivy.

5. Image Processing

Due to the ever-increasing use of Machine Learning, the role of image (pre)processing tools has also skyrocketed. To fulfill this demand, Python offers a host of libraries that are useful for multiple applications like face recognition & image detection. Some of the popular image processing Python libraries include OpenCV and Python Imaging Library(PIL).

6. Text Processing

Text Processing is among the most common uses of Python. Text Processing allows you to handle enormous volumes of text while giving you the flexibility to structure it as you wish. Do you ever think how facebook detects hate speech posts and comments? Well, that has done with Python's text processing capabilities.

7. Audio and Video Applications

When it comes to working with audio and video files, python is fully equipped with tools and libraries to accomplish your task. Tasks such as basic signal processing, creative audio manipulation, audio recognition, and more can be easily handled by libraries like Pyo, pyAudioAnalysis, Dejavu, and many other libraries like it.

As for the video part, Python offers several libraries, such as Scikit-video, OpenCV, and SciPy, that can help you manipulate and prepare videos for use in other applications. Popular audio & video streaming applications like Spotify, Netflix, and YouTube are written in Python.

8. Web Scraping Applications

The internet is home to an enormous amount of information ready to be utilized. What Web Scrapers essentially do is they crawl the websites they're directed towards and store all the collected information from their web pages in one place. From there onwards, this data could be used by researchers, analysts, individuals, organizations for a broad range of tasks. A few examples of the tools behind the Web Scrapers are PythonRequest, BeautifulSoup and Selenium.

9. Data Science and Data Visualization

Data plays a decisive role in the modern world. Why? because it is key to understanding the people and their taste in things around them by gathering and analyzing crucial insights about them. This is what the entire domain of Data Science revolves around. Data Science involves identifying the problem, data collection, data processing, data exploration, data analysis, and data visualization.

The Python ecosystem offers several libraries that can help you tackle your Data Science problems head-on. We have Matplotlib and Seaborn as the most widely used data visualization tools.

10. Scientific and Numeric Applications

AI, ML, and Data Science projects still require intensive computations in the form of linear algebra, high-level mathematical functions, and similar, Python is well equipped for them too. Python helped scientists and researchers conclude countless number-crunching problems and uncover new findings. FreeCAD and Abaqus are some realworld examples of numerical and scientific applications built with Python.

Conclusion

Python is an extremely robust and versatile programming language that is rapidly gaining popularity among developers from various sectors. Its ability to be deployed into virtually any domain is remarkable, thanks to its vast ecosystem of diverse libraries.

Lesson Three : Python Installation

Python IDEs are software applications that provide tools and features to facilitate coding in Python. They offer features like code writing, execution, debugging, making the development process more efficient.

Jupyter Notebook

Jupyter Notebook is an open source web application that you can use to create and share live code. It is a popular IDE for data science and machine learning tasks.

How to install Jupyter Notebook?

- Download Anaconda
    - Visit the link https://www.anaconda.com/download
    - Select Window / Mac
    - Download the current python version
- Open downloaded file
- Click continue
- Click install for me only
- You get notification **'The installation was completed successfully'**
- Click close

How to write and run a code on Jupyter Notebook?

- Open anaconda prompt
- Select desired folder from the given list of directories
- Click new
- Click python 3
- Write your code inside the cells

> Example: Python Code to print Hello World
>
> ```
> print("Hello World")
> ```

- Rename file name
- Click file
- Click save as
- Click file path (This is optional)
- Click save
- Click 'Shift + Enter' to run your code

- Finally you will get an output ‘Hello World’ on next line to your code Else you made a mistake, so recheck it once again.

```
print("Hello World")
```

```
Hello World
```

Key Takeaways

- *Python is a high-level programming language known for its simplicity and readability.*
- *We study Python because it is widely used, has a large community support, and offers numerous applications in various domains.*
- *Python was created by Guido van Rossum, and it was first released in 1991.*
- *Key features of Python include its easy-to-learn syntax, open source, extensive standard library, portable and platform independent.*
- *Python has a wide range of applications, including web & App development, data analysis, machine learning, scientific computing, automation, and scripting, among others.*
- *Jupyter Notebook is a popular IDE for data science and machine learning tasks.*

// END OF CHAPTER TWO

Click here for Chapter Three

CHAPTER THREE

Variables

Objectives

- *What is a variable?*
- *What are the properties of a variable?*
- *How can we create and use variables in Python?*
- *What is keyword?*
- *What operations can be performed on variables?*

Lesson One : Introduction to Variables

What is Variable?

Variable is the name which we give to the memory location which holds some data. With a variable we can store the data, access the data and also manipulate the data.

PS. Variable is a container where you can store a value.

To summarize, a variable is a

- Name
- Refers to a value
- Hold some data
- Name of a memory location

💡 *Let us consider librarians, how could they storeand access ton of books? They just make catalogue of books based on their genre with their reference location. That is how variables also works.*

Properties of a Variable

1. Type:

💡 *What type of data you store in the variable?*

Each data have a type whether it is number word or something and each variable should belong to one of the data types. This is called the data type of the variable.

2. Scope:

💡 *Who can access these data?*

In any programming language, scope refers to the place or limit or boundaries or the part of the program within which the variable is accessible.

3. Value:

💡 *What do you store in the variable?*

Each and every variable in a program holds value of some type which can be accessed or modified during the due flow of the program.

4. Location:

💡 *Where you store the variable?*

As already stated, the data is stored in memory location. The variable is the name which we give to that location. Each memory location has some address associated with it.

5. Lifetime:

💡 *Till when the variable will be available?*

The lifetime of a variable refers to the time period for which the memory location associated with the variable stores the data.

How to create a variable in Python?

In python, to create a variable we just need to specify

- Name of the variable
- Assign value to the variable

> Syntax to create variable:
>
> `name_of_the_variable = value`

Ex. 3.1 (Create a variable)

```
1 age = 21
```

How can I look for my variables?

To see your variable use print() function

> Syntax to print a variable:
>
> `print(name_of_the_variable)`

Ex. 3.2 (Print a variable)

```
1 print(age)
```

```
21
```

What is print() function?

print() is a function that can print the specified message to your computer screen.

> Syntax to print a message / text:
>
> `print("write your message here")`

PS. Don't worry we will discuss more about function later in Function Chapter.

Ex. 3.3 (Print a message / text)

```
1 print("I am John")
```

```
I am John
```

💡 *What if we want to print some message along with our variable?*

> Syntax to print message with a variable:
>
> `print("write your message here",name_of_the_variable)`

Ex. 3.4 (Print text along with a variable)

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

```
My age is 21
```

EXERCISE 3.1

Create a variable "date" and store today's date then print it?

Hint: Output will be like below

Today's date is 6

```
# write your answer code here

```

💡 *What if we want to submit value of a variable outside our code like questionnaire?*

In such scenario we use input() function

What is input function?

input() is a function that can accept values from a user (you).

> Syntax to submit value of a variable:
>
> `name_of_the_variable = input("Ask user to enter a value")`

Ex. 3.5 (Take input from a user)

```
age = input("Enter your age?\n")
print("Your age is",age)
```

```
Enter your age?
19
Your age is 19
```

Note: We can print meaningful text messages along with variables for better understanding.

- Text message we should keep in within double quotes.
- Text message and variable name should be separated by comma symbol.

EXERCISE 3.2

Create a variable "bd" and take input birth date from user then print it?

Hint: Output will be like below

Tell me your birthdate? 6

Your birthdate is 6

```
# write your answer code here

```

Invalid cases for variables

While defining variables in python,

- Variable names should be on the left side.
- Value should be on the right side.
- Violating this will result in syntax error.

Ex. 3.6 (Creating a variable in wrong direction)

```
17 = age
```

```
  Cell In[8], line 1
    17 = age
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

Rules for creating a variable

1. must start with letter or underscore

Ex. 3.7 (Creating a variable with underscore)

```
_age = 23
print(_age)
```

```
23
```

2. cannot start with number

Ex. 3.8 (Creating a variable starts with number)

```
1dollar = 76
```

```
  Cell In[10], line 1
    1dollar = 76
    ^
SyntaxError: invalid decimal literal
```

3. cannot use any special symbol

Ex. 3.9 (Creating a variable with special symbol)

```
1 $dollar = 74
```

```
  Cell In[11], line 1
    $dollar = 74
    ^
SyntaxError: invalid syntax
```

4. can't use space (instead use underscore)

Ex. 3.10 (Creating a variable using space)

```
1 My Age = 24
```

```
  Cell In[12], line 1
    My Age = 24
       ^
SyntaxError: invalid syntax
```

5. are case sensitive

PS. Case sensitive means you can't use lowercase letters in place of uppercase and vice versa.

Ex. 3.11 (Print a variable using wrong case)

```
1 age = 21
2 print(Age)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 2
      1 age = 21
----> 2 print(Age)

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

6. cannot be a keyword.

Lesson Two : Keywords

What is keyword?

Keywords are special reserved words, that have specific meanings and purposes.

All 33 keywords in python contain only alphabet symbols. All of them are in lower case except True, False, and None.

To see all the keywords –

```
1 import keyword
2 keyword.kwlist
```

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

PS. You cannot use these keywords as a variable, else will result in error.

Ex. 3.12 (Creating a variable with keyword name)

```
1 class = 4
```

```
  Cell In[15], line 1
    class = 4
          ^
SyntaxError: invalid syntax
```

Multiple Variables in a Single Line

We can assign multiple variables to multiple values in a single one line. During assignment the variable name on the left hand side should be equal with the values on the right hand side. If not it results in error.

Ex. 3.13 (Assigning multiple variables)

```
1 Sid, age, grade = 4327, 24, 9
2 print(age, grade, Sid)
```

```
24 9 4327
```

Ex. 3.14 (Unequal items on either side)

```
1 Sid, age, grade = 4327, 24
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[17], line 1
----> 1 Sid, age, grade = 4327, 24

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

Ex. 3.15 (Unequal items on either side)

```
1 Sid, age = 4327, 24, 9
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[18], line 1
----> 1 Sid, age = 4327, 24, 9

ValueError: too many values to unpack (expected 2)
```

Single Value for multiple Variables

We can assign a single value to multiple variables simultaneously.

Ex. 3.16 (Assign single value to multiple variables)

```
Sid = age = grade = 10
print(Sid, age, grade)
```

```
10 10 10
```

Lesson Three : Variable Operations

💡 What if we want to know memory location of a variable?

1. Memory Location of a variable

you can get Memory location of a variable using the syntax below:

```
id(variable_name)
```

PS. For sake of simplicity, from now onward MA means Memory Address of a variable.

Ex. 3.17 (Get Memory Location of a variable)

```
dollar = 69
print("1$ = ",dollar)
print("MA of dollar:",id(dollar))
```

```
1$ =  69
MA of dollar: 140719801867176
```

💡 *What if we want to change value of a variable?*

2. Reassigning

you can update / reassign value of a variable using the syntax below:

```
variable_name = new_value
```

PS. While variable is reassigning, its memory location also changed.

Ex. 3.18 (Updating the value of variable)

```
dollar = 69
print("In 2019, $1 =",dollar)
print("MA of dollar in 2019:",id(dollar))
dollar = 72
print("In 2020, $1 = ",dollar)
print("MA of dollar in 2020:",id(dollar))
```

```
In 2019, $1 = 69
MA of dollar in 2019: 140719801867176
In 2020, $1 =  72
MA of dollar in 2020: 140719801867272
```

💡 *What if we want to delete a variable (remove from memory)?*

3. Deleting

you can delete a variable using the syntax below:

```
del variable_name
```

Ex. 3.19 (Deleting a variable)

```
book = 5
del book
print(book)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[22], line 3
      1 book = 5
      2 del book
----> 3 print(book)

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

Finally, What about Constants?

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

> Example – GRAVITY = 9.81

PS. We will discuss more about constants in later Chapter.

EXERCISE 3.3

Create a variable "ap" and accept "apple price" from user, then print user's apple price "ap" and its memory location, then next reassign apple price to 34, again print apple price and its memory location, finally delete variable "ap"?

Hint: Output will be like below

```
1 # write your answer code here
2
```

Key Takeaways

- *Variable is the name which we give to the memory location which holds some data.*
- *Type, Scope, Value, Location and Lifetime are properties of a variable.*
- *print() is a function that can print the specified message to your computer screen.*
- *input() is a function that can accept values from a user (you).*
- *Keywords are special reserved words, that have specific meanings and purposes.*
- *Checking memory location, reassigning and deleting are operations you can perform on variables.*

// END OF CHAPTER THREE

Click here for Chapter Four

CHAPTER FOUR

Data Types

Objectives

- *What is a data type?*
- *What are the built-in data types in Python?*
- *What is comment?*
- *How can we convert the data type of a variable?*

Lesson One : Introduction to Data Types

All programming languages aim is to create some data then do some actions / operations on the data, i.e. processing data. The data can be categorized into different types. Categorizing the data is important in order to perform the operations on it. That classification, which states the type of data stored in the variable is called data type.

What is a data type?

Data Type is type of data that to be stored in a variable, simple as that.

Generally based on their creations data types can be classified into two types:

1. User defined data types: Data types which are created by programmer (you), example is class, module, array etc.

PS. Don't worry, We will discuss it more later in OOPS Chapter.

2. Built-in data types: Data types which are already available in python.

Built-in data types also classified into:

- None
- Bool / booleans
- Numeric types
    - Int
    - Float
    - Complex
- Sequences / Collections
    - list
    - tuple
    - set
    - dict
    - str

💡 *What if we want to create empty variable (a variable without value)?*

None data type

None data type represents an object that does not contain any value. If any object has no value, then we can assign that object with None type.

> Syntax to create None data type variable:
>
> `variable_name = None`

💡 *How to know a variable is belongs to specific data type?*

> Syntax to know data type of a variable:
>
> `type(variable_name)`

What is type() function?

type() is an in built or pre-defined function, which is used to check data type of a variable.

Ex 4.1 (Printing None data type)

```
1 box = None
2 print("The Box is",box)
3 print(type(box))
```

```
The Box is None
<class 'NoneType'>
```

💡 *What if we want to store facts (True / False) values?*

Bool (boolean) data Type

The bool data type represents boolean values in python. bool data type having only two values those are, True and False. Python internally represents, True as 1(one) and False as 0(zero).

Ex 4.2 (Printing bool values)

```
a = True
b = False
print("Gravity = 9.81",a)
print("1 + 2 = 5 is",b)
print(type(a))
print(type(b))
```

```
Gravity = 9.81 True
1 + 2 = 5 is False
<class 'bool'>
<class 'bool'>
```

`EXERCISE 4.1`

`Create a variable "gf" and print any general fact and its data type?`

```
# write your answer code here

```

Lesson Two : Numeric Data Types

💡 *What if we want to store number in a variable?*

1. int data type

The int data type represents values or numbers without decimal values. In python there is no limit for int data type. It can store very large values conveniently.

Ex 4.3 (Print integer value)

```
Sid = 16784
print("My student id is",Sid)
print(type(Sid))
```

```
My student id is 16784
<class 'int'>
```

EXERCISE 4.2

Create a variable "s" and store number of subjects you are taking on this semester then print it and its data type?

```
# write your answer code here

```

💡 *What if we want to store decimal number?*

2. float data type

The float data type represents a number with decimal values.

Ex 4.4 (Print float value)

```
grade = 9.74
print("My grade is",grade)
print(type(grade))
```

```
My grade is 9.74
<class 'float'>
```

EXERCISE 4.3

Create a variable "T" and store current temperature then print it and its data type?

```
# write your answer code here

```

💡 *What if we want to store complex number?*

3. complex data type

The complex data type represents the numbers which is written in the form of a+bj or a-bj, here a is representing a real and b is representing an imaginary part of the number. The suffix small j or upper J after b indicates the square root of -1. The part "a" and "b" may contain integers or floats.

Ex 4.5 (Print complex value)

```
a = 3 + 6j
b = 2 - 5.5j
c = 2.4 + 7j
print(a, b, c)
print(type(a))
```

```
(3+6j) (2-5.5j) (2.4+7j)
<class 'complex'>
```

Lesson Three : Sequence Data Types

Sequences in python are objects that can store a group of values. The below data types are called sequences.

- String
- List
- Tuple
- Set
- Dictionary

💡 *What if we want to store text in a variable?*

1. String data type

A group of characters enclosed within single quotes or double quotes or triple quotes is called a string.

PS. We will discuss more in the String Chapter.

Ex 4.6 (Printing string using single quote)

```
1 name = 'John'
2 print("My name is",name)
3 print(type(name))
```

```
My name is John
<class 'str'>
```

💡 *What if there is an apostrophe in a string?*

Ex 4.7 (Printing string using double quote)

```
1 book = "John's book"
2 print("This is",book)
3 print(type(name))
```

```
This is John's book
<class 'str'>
```

💡 *What if there is both apostrophe and double quote in a string?*

Ex 4.8 (Printing string using triple quote)

```
v = '"John’s book is awesome."'
print("{} Michael J.".format(v))
print(type(v))
```

```
"John’s book is awesome." Michael J.
<class 'str'>
```

`EXERCISE 4.4`

`Create a variable "n", "s", "q" and store your name, your favorite song and your favorite quote respectively. then print it?`

```
# write your answer code here

```

💡 *What if we want to describe our code with title, author, date and moreover about our code?*

What is comment?

Comment is a programmer-readable explanation or annotation in the source code.

Must know about Comment

1. Comments are useful to describe the code in an easy way.
2. Python ignores comments while running the program.
3. To comment python code, we can use hash symbol #
4. For multi-line comments, we can use single / double / triple quotation marks.

Ex 4.9 (Code Description using Comment)

```
# My First Python program
# Author: John Michael
# Date: 21/07/2024
print("Hello World")
```

```
Hello World
```

EXERCISE 4.5

Write any python code that you have learned before then describe your code by

✓ Title of the code

✓ Coder name

✓ Date

✓ Place

```
1 # write your answer code here
2
```

💡 *What if we want to store multiple group of values inside a single variable?*

2. List data type

We can create list data structure by using square brackets [ ]. A list can store different data types. list is mutable (elements can be removed or added).

PS. We will discuss more in the List and Tuple Chapter.

> Syntax to create List :
>
> `name_of_list = [element,...]`

Ex 4.10 (Creating a List)

```
1 detail = [1, 'John', 9.5, 'pass', 1]
2 print("My details",detail)
3 print(type(detail))
```

```
My details [1, 'John', 9.5, 'pass', 1]
<class 'list'>
```

EXERCISE 4.6

Create a list of your favorite subjects then print it and its data type?

```
1 # write your answer code here
2
```

💡 *What if we don't want to add or remove elements (Create unmodifiable data)?*

3. Tuple data type

We can create tuple data structure by using parenthesis (). A tuple can store different data types. tuple is immutable.

PS. We will discuss more in List and Tuple Chapter.

> Syntax to create Tuple :
>
> `name_of_tuple = (element,...)`

Ex 4.11 (Creating a Tuple)

```
1 detail = (1, 'John', 9.5, 'pass', 1)
2 print("My details",detail)
3 print(type(detail))
```

```
My details (1, 'John', 9.5, 'pass', 1)
<class 'tuple'>
```

`EXERCISE 4.7`

`Create a tuple of days in a week then print it and its data type?`

```
1 # write your answer code here
2
```

💡 *What if we don't want repeated element(like 1)?*

4. Set data type

We can create set data structure by using parentheses symbols (). set can store same type and different type of elements.

PS. We will learn more in the Set Chapter.

> Syntax to create Set :
>
> `name_of_set = {element,...}`

Ex 4.12 (Create a Set)

```
1 detail = {1, 'John', 9.5, 'pass', 1}
2 print("My details",detail)
3 print(type(detail))
```

```
My details {'pass', 1, 9.5, 'John'}
<class 'set'>
```

EXERCISE 4.8

Create a set of your favorite books then print it and its data type?

```
# write your answer code here

```

💡 *What if we want to pair of related elements (for example pair of Athlete's name with their rank)?*

5. Dictionary data type

We can create dictionary type by using curly braces {}. dict represents a group of elements in the form of key value pairs like map.

PS. We will discuss more in Dictionary Chapter.

> Syntax to create Dictionary :
>
> `name_of_dictionary = {key:value,...}`

Ex 4.13 (Create Dictionary)

```
Info = {2:'John', 1:'Mike', 3:'Dani'}
print("Race",Info)
print(type(Info))
```

```
Race {2: 'John', 1: 'Mike', 3: 'Dani'}
<class 'dict'>
```

EXERCISE 4.9

Create a football player and jersey pair of dictionary then print it and its data type?

```
# write your answer code here

```

Lesson Four : Conversion of Data Types

Based on requirements, developers/programmers can convert from one data type into another data type explicitly, this is called type conversion. Python provides the following inbuilt functions to convert from one type to another type.

- int() : convert from other type into int type

- float() : convert from other type into float type
- complex() : convert from other type to complex
- bool() : convert from other type into bool type
- str() : convert from other type into str type

1. Convert to float

Ex 4.14 (Convert to float data type)

```
i = 65
itf = float(i) #Integer to float
print(itf)
print(type(itf))
```

```
65.0
<class 'float'>
```

2. Convert to integer

Ex 4.15 (Convert float to integer data type)

```
f = 65.89
fti = int(f) #float to Int
print(fti)
print(type(fti))
```

```
65
<class 'int'>
```

Ex 4.16 (Convert string to integer data type)

```
st = '123'
sti = int(st, 10) #Str(No.) to Int
print(sti)
print(type(sti))
```

```
123
<class 'int'>
```

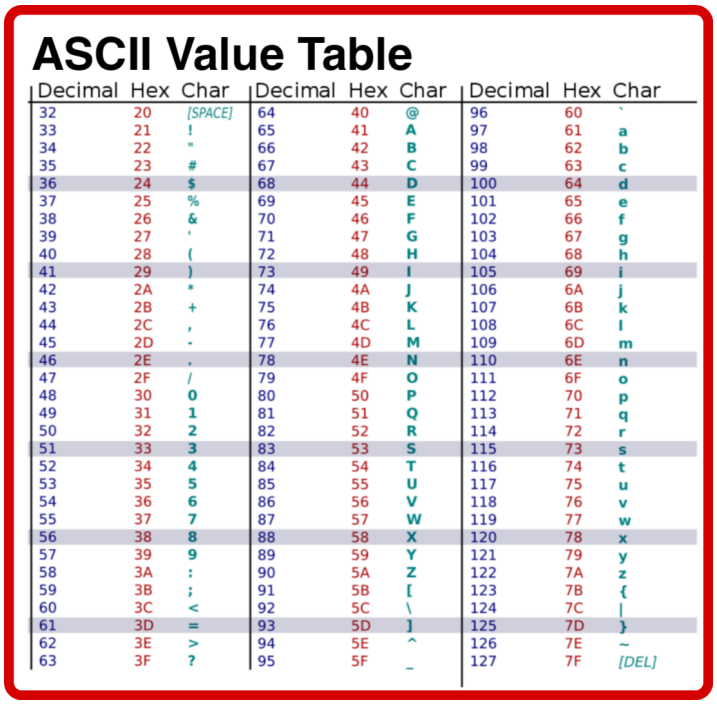
Ex 4.17 (Convert character to integer data type)

```
ch = 'A' #Character to Int(ASCII)
cti = ord(ch)
print(cti)
print(type(cti))
```

```
65
<class 'int'>
```

What is ASCII Value?

ASCII Value is a standard that assigns letters, numbers, and other characters.



3. Convert to complex

Ex 4.18 (Convert Real No. to complex data type)

```
r = 6 #Real No. to Complex No.
rtc = complex(r)
print(rtc)
print(type(rtc))
```

```
(6+0j)
<class 'complex'>
```

4. Convert to string

Ex 4.19 (Convert integer to string data type)

```
i = 65
itst = str(i) #Int to Str
print(itst)
print(type(itst))
```

```
65
<class 'str'>
```

Ex 4.20 (Convert ASCII to string data type)

```
1 nta = chr(65) #ASCII Value to str
2 print(nta)
3 print(type(nta))
```

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

5. Convert to list

Ex 4.21 (Convert Tuple to List data type)

```
1 t = (1, 'John', 9.5, 'pass')
2 ttl = list(t) #Tuple to List
3 print(ttl)
4 print(type(ttl))
```

```
[1, 'John', 9.5, 'pass']
<class 'list'>
```

Ex 4.22 (Convert Set to List data type)

```
1 s = {1, 'John', 9.5, 'pass'}
2 stl = list(s) #Set to List
3 print(stl)
4 print(type(stl))
```

```
['pass', 1, 9.5, 'John']
<class 'list'>
```

6. Convert to tuple

Ex 4.23 (Convert List to Tuple data type)

```
1 l = [1, 'John', 9.5, 'pass']
2 ltt = tuple(l) #List to Tuple
3 print(ltt)
4 print(type(ltt))
```

```
(1, 'John', 9.5, 'pass')
<class 'tuple'>
```

Ex 4.24 (Convert Set to Tuple data type)

```
s = {1, 'John', 9.5, 'pass'}
stt = tuple(s) #Set to Tuple
print(stt)
print(type(stt))
```

```
('pass', 1, 9.5, 'John')
<class 'tuple'>
```

7. Convert to set

Ex 4.25 (Convert to Set data type)

```
l = [1, 'John', 9.5, 'pass']
lts = set(l) #List to Set
print(lts)
print(type(lts))
t = (1, 'John', 9.5, 'pass')
tts=set(t) #Tuple to Set
print(tts)
print(type(tts))
```

```
{'pass', 1, 9.5, 'John'}
<class 'set'>
{'pass', 1, 9.5, 'John'}
<class 'set'>
```

Key Takeaways

- *Data Type is type of data that to be stored in a variable.*
- *type() is a function, which is used to check data type of a variable.*
- *Python's built-in data types include None, boolean, numeric, and sequence types.*
- *Comment is a programmer-readable explanation or annotation in the source code.*

// END OF CHAPTER FOUR

Click here for Chapter Five

CHAPTER FIVE

Operators

Objectives

- *What is an operator?*
- *Which operators you can use in Python?*
- *What is Assignment Operator?*
- *What are Arithmetic Operators?*
- *What is Compound Operator?*
- *What are Relational Operators?*
- *What is Identity Operator?*

Lesson One : Introduction to Operators

Computer is a device that can accept input data from a user and process the given data to produce a desired output. Nevertheless processing data means doing some sort of calculations on the given inputs to give output back to the user. Programming is more than storing data in a variable, operators allow us to do exciting things with the data.

What is an operator?

An operator is a symbol which is applied on some operands (usually variables), to perform certain action or operation. For example,

```
a = 1
b = 2
c =  a + b
print(c)

Output: 3
```

In the above example, first we assigned values to two variables ‘a’ and ‘b’ and then we added both the variables and stored the result in variable ‘c’.

Note: The symbols + and = are called operators and the variables a,b,c on which operation is being performed are called operands.

Basic Classification of Operators

1. Unary Operator – If the operator acts on a single operand then it’s called unary operator.

💡 *What if you want to make a positive number negative?*

For example, in ‘-5’ the operator ‘-’ is used to make the (single) operand ‘5’ a negative value.

2. Binary Operator – If the operator acts on two operands then it’s called a binary operator.

💡 *What if you want to add numbers?*

For example, ‘+’ operators need two variables (operands) to perform addition operation.

3. Ternary Operator – If the operator acts on three operands then it’s called a ternary operator.

PS. Don’t panic, We will see more in detail on upcoming chapters (if … else Statement).

Types of Operators in Python:

1. Unary Minus Operator
2. Assignment Operator
3. Arithmetic Operators
4. Compound Operators
5. Relational Operators
6. Identity Operators
7. Logical Operators
8. Bitwise Operator
9. Membership Operators

💡 *What if we want to make a positive number negative? Yes we can, How? Let us see…*

Unary Minus Operator (-)

The operator ‘-’ is called Unary minus operator. It is used to negate the number.

This operator operates on a single operand, hence unary operator. This is used to change a positive number to a negative number and vice-versa.

Ex 5.1 (Unary Minus Operator)

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

```
10
-10
```

Earlier, When we discuss about creating a variable we have used '=' let us recall

> `age = 21`

💡 *What is '=' ?*

Lesson Two : Assignment Operators

What is Assignment Operator?

Assignment Operators are operators which are used for assigning values to variables. '=' is the assignment operator.

💡 *What if we want copy of a variable?*

> Syntax :
> `new_variable = old_variable`

PS. Memory Address of both new (copied) and old variables is unique.

Ex 5.2 (Copy variable using Assignment Operator)

```
v = 10
c = v
print("Old variable is", v)
print("Copied variable is", c)
print("Memory Address of Old variable:", id(v))
print("Memory Address of Copy variable", id(c))
```

```
Old variable is 10
Copied variable is 10
Memory Address of Old variable: 137772376572432
Memory Address of Copy variable 137772376572432
```

EXERCISE 5.1

Create a string variable "name" and store your name then copy it to a "new" variable, after that print both variables and their memory location?

```
# write your answer code here

```

💡*What if we want to add two numbers? Yes we can, How? Let us see…*

Lesson Three : Arithmetic Operations

What are Arithmetic Operators?

Arithmetic Operators are operators which are used to perform basic mathematical calculations like addition, subtraction, multiplication, division etc. They are: +, -, , /, %, *, //

Note:

- Division operator / always performs floating point arithmetic, so it returns float value.
- Floor division (//) can perform both floating point and integral as well,
- If values are int type, then result is int type.
- If at least one value is float type, then the result is of float type.

Ex 5.3 (Arithmetic Operators)

```
a = 20
b = 12
print(a + b) #Addition
print(a - b) #Subtraction
print(a / b) #Division
print(a // b) #Quotient
print(a % b) #Remainder
print(a * b) #Multiplication
print(a ** b) #Square
print(a - b + 3 * 4 / 3) #BODMAS
```

```
32
8
1.6666666666666667
1
8
240
4096000000000000
12.0
```

EXERCISE 5.2

Create two integer variables "a" and "b" accept values from a user then calculate their average?

```
# write your answer code here

```

💡*What if we want perform complex calculations like logarithm, trigonometric, exponential etc…?*

math() Function

> Syntax for math function:
>
> `from math import *`
>
> `function(value)`

Ex 5.4 (Complex calculations using math function)

```
from math import *
print(log(10)) #log base e
print(log10(10)) #log base 10
print(sin(90)) #sin of radian
print(radians(90)) #radian to deg
print(sin(radians(90))) #sin of deg
print(pow(2,3)) #2 power of 3
print(exp(2)) #e power of 2
print(sqrt(16)) #square root
print(factorial(5)) #factorial
print(ceil(4.6)) #largest integer
print(floor(4.6)) #smallest integer
print(e) #constant e
print(pi) #constant pi
```

```
2.302585092994046
1.0
0.8939966636005579
1.5707963267948966
1.0
8.0
7.38905609893065
4.0
120
5
4
2.718281828459045
3.141592653589793
```

EXERCISE 5.3

Calculate

- **ln(e)**
- **cos(90°)**
- **10 the power of 5**
- **Square root of 625**
- **Ceil and Floor of 6.5**

using math() function?

```
# write your answer code here

```

💡 *What if we want to increase value of a variable? Yes we can, How? Let us see*

Ex 5.5 (When you deposit on your bank account)

```
balance = 12345
print("Balance before deposit:",balance)
deposit = 2655
balance = balance + deposit
print("Balance after deposit:",balance)
```

```
Balance before deposit: 12345
Balance after deposit: 15000
```

💡 *Why we write a variable twice? We can write it only once, How? Let us see…*

Lesson Four : Compound Operators

What is Compound Operator?

Compound Operators are operators which is combination of some arithmetic and assignment operators (+=, -=, =, /=, %=, *=, //=).

> Syntax to compound operators:
>
> `result_v operator = operand_v`

`balance = balance + deposit` is same as `balance += deposit`

That means add "balance" and "deposit" then store the result in variable "balance".

Ex 5.6 (Addition using Compound Operator)

```
balance = 12345
print("Balance before deposit:",balance)
deposit = 2655
balance += deposit
print("Balance after deposit:",balance)
```

```
Balance before deposit: 12345
Balance after deposit: 15000
```

Ex 5.7 (Compound Operators)

```
a, b = 2, 3
a+=b #Addition Assignment
print(a)
a-=b #Subtraction Assignment
print(a)
a*=b #Multiplication Assignment
print(a)
a//=b #Quotient Assignment
print(a)
a%=b #Remainder Assignment
print(a)
a**=b #Square Assignment
print(a)
```

```
5
2
6
2
2
8
```

EXERCISE 5.4

Create two integer variables "avg" and "b" accept from user then calculate their average using compound operators?

```
# write your answer code here

```

💡 *What if we want to compare two numbers or variables?*

Lesson Five : Relational Operators

What are Relational Operators?

Relational Operators are operators which are used to check for some relation like greater than or less than etc.. between operands.

They are: <, >, <=, >=, ==, !=

These are used to compare two values for some relation and return booleans (True or False) depending the relation.

Ex 5.8 (Relational Operators)

```
print(1 == 2) #Equal
print(1 != 2) #Not Equal
print(1 > 2) #Less Than
print(1 < 2) #Greater Than
print(1 <= 2) #Less Than Or Equal
print(1 >= 2) #Greater Than Or Equal
```

```
False
True
False
True
True
False
```

EXERCISE 5.5

Create two variables "my_age" and "friend_age" accept from user then compare them?

```
# write your answer code here

```

💡 *What if you want to compare address of two variables?*

Lesson Six : Identity Operators

What is Identity Operator?

Identity Operators are operators which are used to check for identity by comparing the memory location (address) of two variables. With these operators, we will be able to know whether the two variables are pointing to the same location or not.

Recall that : The memory location of the object can be seen using id() function.

There are two identity operators, is and is not.

1. is:

- A is B returns True, if both A and B are pointing to the same address.
- A is B returns False, if both A and B are not pointing to the same address.

2. is not:

- A is not B returns True, if both A and B are not pointing to the same object.
- A is not B returns False, if both A and B are pointing to the same object.

Ex 5.9 (Identity Operators)

```
a, b, c = 25, 25, 30
print(a is b)
print(a is c)
print(id(a))
print(id(b))
print(id(c))
```

```
True
False
140719801865768
140719801865768
140719801865928
```

EXERCISE 5.6

Create two variables "a" and "b" then compare them?

```
# write your answer code here

```

Note: Identity operators are not comparing values of objects. They compare memory address of objects. If we want to compare value of objects, we should use relational operator '==’.

Ex 5.10 (Identity Operators Vs Relational Operator)

```
a = []
b = []
c = a
print(id(a))
print(id(b))
print(id(c))
print(a == b)
print(a is b)
print(a is c)
print(a == c)
```

```
2091128894208
2091128891264
2091128894208
True
False
True
True
```

II Pause the next operators for a while

Key Takeaways

- *An operator is a symbol used to perform actions or operations on variables.*
- *There are 9 types of operators supported by python.*
- *Assignment operators assign values to variables using the '=' symbol.*
- *Arithmetic operators perform basic mathematical calculations such as +, -, x, and %.*
- *Compound operators combine arithmetic and assignment operators (+=, -=, =, /=, %=, *=, //=).*
- *Relational operators check relations between operands, such as greater than or less than.*
- *Identity operators compare the memory location (address) of two variables to check for identity.*

// END OF CHAPTER FIVE

Click here for Chapter Six

CHAPTER SIX

Conditional Statement

Objectives

- *What is Conditional Statement?*
- *What is Indentation?*
- *What are Logical Operators?*
- *What is nested if statement?*
- *What is Bitwise Operator?*
- *What is Membership Operators?*
- *What is eval() function?*

Lesson One : Introduction to Conditional Statements

Decision Making in programming is similar to decision making in real life. In programming, a certain block of code needs to be executed when some condition is fulfilled.

What is Conditional Statement?

Conditional /decision making/ Statements are statements executed based on the condition.

Python Interpreter execute sequentially when there is no condition around the statements. If you put some condition for a block of statements, the execution flow may change based on the result evaluated by the condition.

There are four types of conditional statements in python. They are as follows:

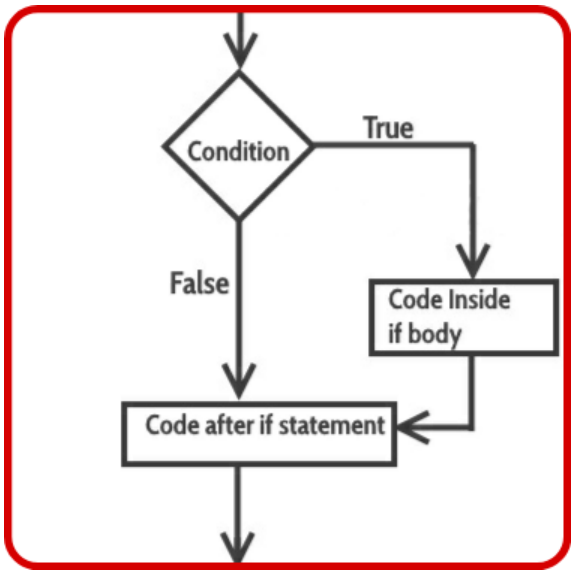
1. if statement
2. if else statement
3. Nested if statement
4. if elif else statement

Lesson Two : if Statement

The if statement is used to test a specific condition. If the condition is true, if-block statement will be executed.

The condition of if statement can be any valid logical expression which can be either evaluated to True or False.

> Syntax for if statement:
> `if (condition):`
> `    if block statement`
> `next line of code`



💡 *Why do we have these much space before if-block statement?*

In Python, indentation is used to separate the if-block statement from other block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation.

What is Indentation?

Indentation is the spaces at the beginning of a code line to indicate a block of code.

Note: if statement contains an expression/condition. As per the syntax colon (:) is mandatory otherwise it throws syntax error.

Condition gives the result as a bool type, either True or False. If the condition result is True, then if block statements will be executed. If the condition result is False, then next line of code execute.

💡 *Suppose you want hire employees for your company and your main condition is employee's age should be above 18 because hiring under age is prohibited. let us develop a code to filter job applicants.*

Ex 6.1 (if statement)

```
a = int(input("Enter your age\n"))

if a > 17:
    print("Pass for interview.")
```

```
Enter your age
12
```

Ex 6.2 (Indentation Error)

```
a = int(input("Enter your age\n"))

if a > 17:
print("Pass for interview.")
```

```
  File "<ipython-input-8-31238eecf411>", line 4
    print("Pass for interview.")
    ^
IndentationError: expected an indented block after 'if' statement on line 3
```

EXERCISE 6.1

Create a variable "age" and take input from user then check vote eligibility (age > 18)?

```
# write your answer code here

```

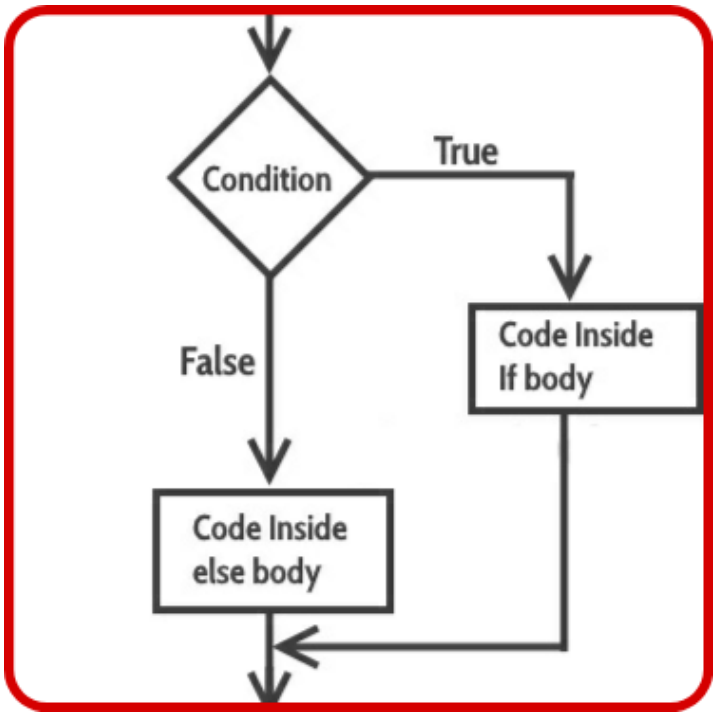
💡 *What if you want to reject applicants who could not fulfill the age condition?*

PS. Clear rejection is better than ignorance.

Lesson Three : if else statement

If the condition result is True, then if block statements will be executed. If the condition result is False, then else block statements will be executed.

> Syntax for if else statement:
> `if (condition):`
> `    if block statement`
> `else:`
> `    else block statement`
> `next line of code`



Ex 6.3 (if else statement)

```
1 a = int(input("Enter your age\n"))
2
3 if a > 17:
4     print("Pass for interview.")
5 else:
6     print("Application Denied.")
```

```
Enter your age
12
Application Denied.
```

`EXERCISE 6.2`

`Create a variable "username" and its value is your name then create log in conditional statement based on username?`

`Hint: Output looks like below`

`Enter your username`

`john`

`Welcome to facebook.`

```
# write your answer code here

```

💡 *Here is a problem, What if a 90 years old man applied for the job, will you accept him? Hell No*

Therefore we need to add additional condition to specify required age range.

► Resume the next operators for a while

💡 *What if we want to combine two relational operators. How? Let us see…*

In such scenarios, we use Logical Operators

What are Logical Operators?

Logical Operators are operators which do some logical operation on the operands and return True or False.

In python, there are three types of logical operators. They are and, or, not. These operators are used to construct compound conditions, combinations of more than one simple condition. Each simple condition gives a boolean value which is evaluated, to return final boolean value.

Note: In logical operators, False indicates 0(zero) and True indicates non-zero value. Logical operators on boolean types

- **and** : If both the arguments are True then only the result is True
- **or** : If at least one argument is True then the result is True
- **not** : complement of the boolean value

Ex 6.4 (Logical operators on boolean types)

```
a, b = True, False

print(a and b) # and
print(a or b) # or
print(not a) # not
```

```
False
True
False
```

II Pause the next operators for a while

Ex 6.5 (if else statement with compound condition)

```
a = int(input("Enter your age\n"))

if a > 17 and a < 45:
    print("Pass for interview.")
else:
    print("Application Denied.")
```

```
Enter your age
90
Application Denied.
```

EXERCISE 6.3

Create two variable and "nationality" and "cgpa" and store values is your name and nationality, then create scholarship conditional statement based on nationality and cgpa?

Hint: Output looks like below

Enter your nationality

Indian

Enter your cgpa

8.7

Congratulations, you won.

```
# write your answer code here

```

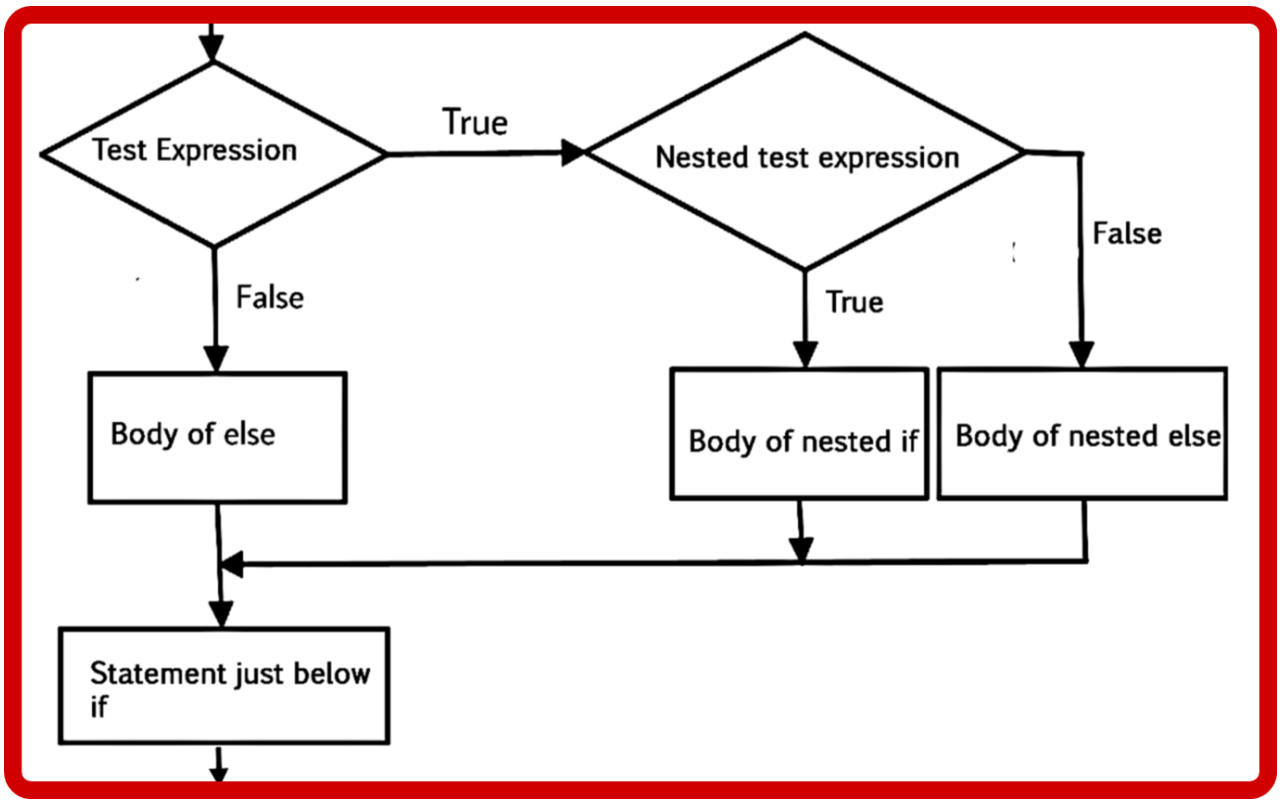
💡 *What if you have more additional condition like (Applicant should have GPA above than 9 else he/she will be in waiting list) ?*

PS. Quality is determined by accuracy and completeness.

Lesson Four : Nested if Statement

Nested if statement is an if statement inside another if statement. Here, a user can decide among multiple options based on multiple conditions.

> Syntax for Nested if statement:
>
> ```
> if (condition 1):
>     if (condition 2):
>         if block statement of condition 2
>     else:
>         else block statement of condition 2
> else:
>     else block statement of condition 1
> next line of code
> ```



Ex 6.6 (Nested if statement)

```
a = int(input("Enter your age\n"))
c = float(input("Enter your GPA\n"))

if a > 17 and a < 45:
    if c > 9:
        print("Accepted.")
    else:
        print("Waiting List...")
else:
    print("Application Denied.")
```

```
Enter your age
28
Enter your GPA
8.5
Waiting List...
```

EXERCISE 6.4

Create two variable "username" and "password" and store values is your name and password, then create log in conditional statement based on username and password?

Hint: Output looks like below

Enter your username

john

Enter your password

12345

Welcome to facebook.

```
# write your answer code here

```

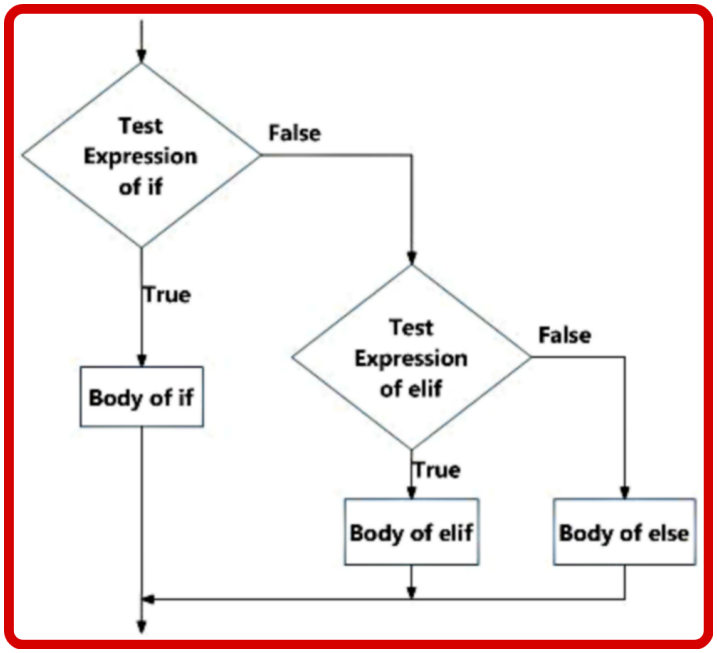
💡 *Now, you couldn't get a person who have GPA more than 9, and the job vacancy is urgent, so you need to reduce GPA criteria to lower range.*

Lesson Five : if elif else Statement

The conditions are evaluated in the written order. If one condition becomes True then the set of statements associated with that particular condition are executed and the execution comes out without checking other conditions.

Syntax for if elif else statement:

```
if (condition):
    if block statement
elif:
    elif block statement
else:
    else block statement
next line of code
```



Ex 6.7 (if elif else statement)

```
1 a = int(input("Enter your age\n"))
2 c = float(input("Enter your GPA\n"))
3
4 if a > 17 and a < 45 and c > 9:
5     print("Accepted.")
6 elif a > 17 and a < 45 and c > 7 and c < 9:
7     print("Pass for interview.")
8 else:
9     print("Application Denied.")
```

```
Enter your age
21
Enter your GPA
8.5
Pass for interview.
```

EXERCISE 6.5

Create a variable "score" and take input from user, then classify them based on their score?

Classification:

"A" – 90 to 100

"B" – 80 to 90

"C" – 70 to 80

"D" – 60 to 70

"F" – below 60

```
# write your answer code here

```

Lesson Six : More Examples on Conditional Statements

Ex 6.8 (if else statement)

```
a = float(input("Enter 1st No: "))
b = float(input("Enter 2nd No: "))

if a > b:
    print(a,"is the largest No.")
else:
    print(b,"is the largest No.")
```

```
Enter 1st No: 45
Enter 2nd No: 21
45.0 is the largest No.
```

> **Recall** Syntax for if else using Tertiary Operator:
> `[on_True] if (condition) else [on_False]`

Ex 6.9 (if else statement using tertiary operator)

```
a = float(input("Enter 1st No: "))
b = float(input("Enter 2nd No: "))

max = a if a > b else b

print("Largest No.=", max)
```

```
Enter 1st No: 45
Enter 2nd No: 21
Largest No.= 45.0
```

Ex 6.10 (if elif else statement)

```
a = float(input("Enter 1st No: "))
b = float(input("Enter 2nd No: "))

if a > b:
    print(a,"is the largest No.")
elif a == b:
    print(a,"and",b,"are equal.")
else:
    print(b,"is the largest No.")
```

```
Enter 1st No: 45
Enter 2nd No: 45
45.0 and 45.0 are equal.
```

Ex 6.11 (Half Nested if statement)

```
a = float(input("Enter 1st no: "))
b = float(input("Enter 2nd no: "))
c = float(input("Enter 3rd no: "))

if a > b:
    if a > c:
        print("largest no.= ",a)
    else:
        print("largest no.= ",c)
else:
    print("largest no.= ",b)
```

```
Enter 1st no: 21
Enter 2nd no: 45
Enter 3rd no: 13
largest no.=  45.0
```

💡 *What if a and b are equal numbers. For example: (1, 1, 2) or (1, 2, 2) or (2, 1, 2) (2, 2, 2)*

Ohh.. Hell no it is incorrect output, Here we go to fix it because we are programmers.

Ex 6.12 (Full Nested if statement)

```
a = float(input('Enter 1st No: '))
b = float(input('Enter 2nd No: '))
c = float(input('Enter 3rd No: '))

if(a >= b):
    if(a >= c):
        largest = a
    else:
        largest = c
else:
    if(b >= c):
        largest = b
    else:
        largest = c

print("The largest No is",largest)
```

```
Enter 1st No: 21
Enter 2nd No: 21
Enter 3rd No: 45
The largest No is 45.0
```

► Resume the next operators for a while

💡 *What if we want to swap two variables?*

Variable Swapping

- Using third variable
- Using Assignment operator
- Using * and / operators
- Using ^ (XOR) operator

Ex 6.13 (Using third variable)

```
a, b = 1, 2

print("Before Swap a=",a,"b=",b)
temp = a
a = b
b = temp
print("After Swap a=",a,"b=",b)
```

```
Before Swap a= 1 b= 2
After Swap a= 2 b= 1
```

Ex 6.14 (Using Assignment operator)

```
a, b = 1, 2

print("Before Swap a=",a,"b=",b)
a, b = b, a
print("After Swap a=",a,"b=",b)
```

```
Before Swap a= 1 b= 2
After Swap a= 2 b= 1
```

Ex 6.15 (Using + and – operators)

```
a, b = 1, 2

print("Before Swap a=",a,"b=",b)
a = a + b
b = a - b
print("After Swap a=",a,"b=",b)
```

```
Before Swap a= 1 b= 2
After Swap a= 3 b= 1
```

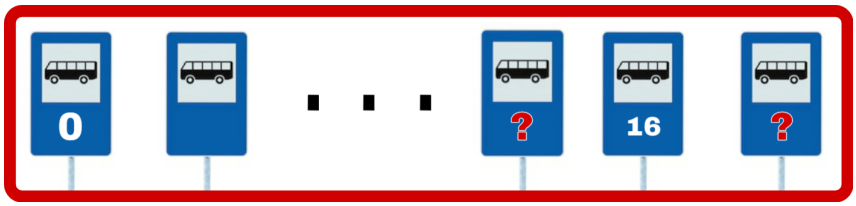
Ex 6.16 (Using * and / operators)

```
a, b = 1, 2

print("Before Swap a=",a,"b=",b)
a = a * b
b = a / b
a = a / b
print("After Swap a=",a,"b=",b)
```

```
Before Swap a= 1 b= 2
After Swap a= 2.0 b= 1.0
```

Case Study : Bus Stop Problem



There are bus stops in a road each bus stop have its own number. Bus station (initial) number is 0. Each bus stop number is double of previous bus stop number. Current bus stop number is 16,

- What is the bus stop number before the current bus stop?
- What is the bus stop number next to the current bus stop?

Ohhh… that is so simple. Okay let us do with a code

Ex 6.17 (Bus Stop Problem by Arithmetic operators)

```
c = int(input("enter current stop\n"))

p = c / 2
p = int(p)
n = c * 2

print("Previous bus stop is",p)
print("Current bus stop is",c)
print("Next bus stop is",n)
```

```
enter current stop
16
Previous bus stop is 8
Current bus stop is 16
Next bus stop is 32
```

💡 *We can also do the previous Bus Stop Problem using another operator called Bitwise Operators. Let us see…*

Lesson Seven : Bitwise Operators

What is Bitwise Operator?

Bitwise Operators are operators used to perform bitwise calculations on integers.

Note : Bitwise operators work only on integers.

There are six bitwise operators, those are

- Bitwise AND
- Bitwise OR
- Bitwise NOT
- Bitwise XOR
- Bitwise right shift
- Bitwise left shift

Ex 6.18 (Bus Stop Problem using bitwise operators)

```
c = int(input("Enter current stop\n"))

p = c >> 1
n = c << 1
print("Previous bus stop is",p)
print("Current bus stop is",c)
print("Next bus stop is",n)
```

```
Enter current stop
16
Previous bus stop is 8
Current bus stop is 16
Next bus stop is 32
```

Ex 6.19 (Swapping Using ^ (XOR) operator)

```
a = 1
b = 2

print("Before Swap a=",a,"b=",b)
a = a ^ b
b = a ^ b
a = a ^ b
print("After Swap a=",a,"b=",b)
```

```
Before Swap a= 1 b= 2
After Swap a= 2 b= 1
```

SUMMARY OF BITWISE (for binary numbers)

x	y	x&y	x|y	x^y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

💡 *What if you want to search a keyword in a text?*

Lesson Eight : Membership Operators

What is Membership Operators?

Membership operators are operators which are used to check whether an element is present in a sequence of elements or not. Here, the sequence means strings, list, tuple, set and dictionaries.

There are two membership operators. i.e.

1. in operator: The in operators returns True if element is found in the collection of sequences. returns False if not found.

2. not in operator: The not in operator returns True if the element is not found in the collection of sequence. returns False in found.

Ex 6.20 (Membership Operators)

```
text = "Welcome"

print("Well" in text)
print("wel" in text)
print("Wel" in text)
print("come" not in text)
```

```
False
False
True
False
```

💡 *What if you want to check a person whether he/she is belong to class 10A or not?*

Ex 6.21 (Membership Operators)

```
class_10A = ["John", "Eden", "Bob"]

print("john" in class_10A)
print("Lucas" not in class_10A)
```

```
False
True
```

💡 *What if a user just want to evaluate some calculation without using variables?*

Lesson Nine : eval() Function

This is an in-built function available in python, which takes the strings as an input. The strings which we pass to it should, generally, be expressions. The `eval()` function takes the expression in the form of string and evaluates it and returns the result.

Ex 6.22 (eval() function /Arithmetic Operator/)

```
1 eval('10 + 10')
```

```
20
```

`EXERCISE 6.6`

`Create two integer variables "a" and "b" take input from a user, then calculate their average by using eval() function?`

```
1 # write your answer code here
2
```

💡 *What if we want to take input from user?*

Ex 6.23 (eval() function /Take inputs from a user/)

```
1 eval(input("Enter expression: "))
```

```
Enter expression: 20 - 12 + 3 * 4 / 3
12.0
```

PS. The `eval()` function is not applicable to compound operators.

Ex 6.24 (eval() function /Unary Minus Operator/)

```
1 a = 5
2 eval('-a ')
```

```
-5
```

Ex 6.25 (eval() function /Relational Operator/)

```
1 eval('1 > 2')
```

```
False
```

Ex 6.26 (eval() function /Logical Operator/)

```
1 eval('0 and 1')
```

```
0
```

Ex 6.27 (eval() function /Bitwise Operator/)

```
1 eval('2 << 3')
```

```
16
```

Ex 6.28 (eval() function /Membership Operator/)

```
1 s = "John"
2 eval('"Jo" in s')
```

```
True
```

Ex 6.29 (eval() function /Identity Operator/)

```
1 eval('1 is 2')
```

```
<string>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
False
```

Ex 6.30 (eval() function /math() function/)

```
1 from math import *
2
3 print(eval('log(1)'))
```

```
0.0
```

EXERCISE 6.7

Calculate

- ln(e)
- cos(90°)
- 10 the power of 5
- Square root of 625
- Ceil and Floor of 6.5

```
1 # write your answer code here
2
```

Key Takeaways

- *Conditional Statements are statements executed based on the condition.*
- *Indentation is the spaces at the beginning of a code line to indicate a block of code.*
- *Logical Operators are operators which do some logical operation on the operands and return True or False.*
- *Nested if statement is an if statement inside another if statement.*
- *Bitwise Operators are operators used to perform bitwise calculations on integers.*
- *Membership operators are operators which are used to check whether an element is present in a sequence of elements or not.*
- *The eval() function takes the expression in the form of string and evaluates it and returns the result.*

// END OF CHAPTER SIX

Click here for Chapter Seven

CHAPTER SEVEN

Function

Objectives

- *What is a function and why is it necessary?*
- *When should we use functions?*
- *What are the types of functions in Python?*
- *What is an argument?*
- *What is a nested function?*
- *What is a recursive function?*
- *What are global and local variables in a function?*
- *What are lambda functions?*
- *What are decorators and generators?*

Lesson One : Introduction to Function

Why function is required?

💡 *Let us understand this with an example. When you go to school in every morning, the things you do are*

1. *Get up from the bed*
2. *Wash your face*
3. *Brush your teeth*
4. *Wear your uniform*
5. *Eat your breakfast*
6. *Go to School.*

Think of this sequence of steps to do morning to go to school. Now when your mom wakes you up for school, she doesn't need to explain all these steps each time to you. Whenever your mom says, "Get ready for school", it's like making a function call. "Going to school" is an abstraction for all the above steps involved.

When should we go for function?

While writing coding logic, instead of writing like a plain text, it's good to keep those coding statements in one separate block, because whenever required then we can call these. If a group of statements is repeatedly required, then it is not recommended to write these statements each and every time separately.

So, it's good to define these statements in a separate block. After defining a function we can call directly if required. This block of statements is called a function. Let us understand more by doing practically.

What is a Function?

A function is a group of statements or a block of code to perform a certain task.

NB: Function is a block of code which only runs when it is called.

The advantages of using functions are:

- Maintaining the code is an easy way.
- Organize and manage our code
- Code re-usability.

> Example: `print()` is predefined function in python which prints output on the desktop.

Types of functions

There are many categories based on which we can categorize the functions.

PS. This categorization is based on who created it.

1. Pre-defined or built-in functions
2. User-defined functions

Predefined or built-in functions

The functions which come installed along with python software are called predefined or built-in functions. We have covered some inbuilt functions in examples of earlier chapters. Some of them are `id(), type(), input(), print()` etc.

User-defined functions

The functions which are defined by the developer as per the requirement are called as user defined functions. In this chapter we shall concentrate on these kind of functions which are used defined.

How to Create and call a Function?

From creating a function to using it, these are the two things that are done.

- Defining function
- Calling function

1. Defining a function

> Syntax to define function
>
> ```
> def function_name(parameters):
>     // body of function
>     return value
> ```

- **def keyword** – Every function in python should start with the keyword 'def'. In other words, python can understand the code as part of a function if it contains the 'def' keyword only.
  PS. 'def' means define a function.

- **Name of the function** – Every function should be given a name, which can later be used to call it.
- **Parenthesis** – After the name '()' parentheses are required
- **Parameters** – The parameters, if any, should be included within the parenthesis.
- **Colon symbol ':'** should be mandatorilyplaced immediately after closing the parentheses.
- **Body of function** – All the code that does some operation should go in the body of the function. The body of the function should have indentation of one level with respect to the line containing the 'def' keyword.
- **Return value /statement/**– Return statement should be in the body of the function. It's not mandatory to have a return statement.

Ex 7.1 (Define a function)

```
1 # defining a function
2 def add():
3     a, b = 1, 2
4     c = a + b
5     return c
```

Note: After defining a function we can call the function using its name. While calling a function, we need to pass the parameters if it has any.

2. Calling a function

In the above example we defined a function with the name ‘add’ to add two nos. But when we execute the above code, it will not display any output, because the function is not called. Hence, function calling is also important along with function definition.

Note: When a function is called once then it will execute once, if called twice then it will be executed twice and so on.

> Syntax to call a function:
> `function_name(parameters)`

Ex 7.2 (Calling a function)

```
# calling function
add()
```

```
3
```

While calling the function, we have to call with the same name of the function which we used while defining it, otherwise we will get an error.

Ex 7.3 (Define a function & call it with different name)

```
# defining a function
def add():
    a, b = 1, 2
    c = a + b
    return c

# calling function
two()
```

```
---------------------------------------------------------
--------------------
NameError                                 Traceback
(most recent call last)
<ipython-input-8-e7cdd6ebe0b2> in <cell line: 2>()
      1 # calling function
----> 2 two()

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

EXERCISE 7.1

Create a function to calculate average of two numbers?

```
# write your answer code here

```

What is return value?

As mentioned above, the return statement is included in the function body, and it returns some result after doing the operations.

Some points about return statement

- return is a keyword in python.
- By using return, we can return the result.
- It is not mandatory for a function to have a return statement.
- If there is not return statement in the function body then the function, by default, returns None

EXERCISE 7.2

Create a function to calculate "a power of b" by using eval() function?

```
# write your answer code here

```

How to return multiple values?

In python, a function can return multiple values.

Ex 7.4 (Function which return multiple values)

```
# defining a function
def cal():
    a, b = 3, 2
    c = a + b
    d = a - b
    return c, d

# calling function
cal()
```

```
(5, 1)
```

EXERCISE 7.3

Create a function to calculate (return) sum and average of two numbers?

```
# write your answer code here

```

Based on the parameters, functions can be categorized into two types. They are:

- Function without parameters (Non-parametric Function)
- Function with parameters (Parametric Function)

What is Non-parametric function?

A function which has no parameters in the function definition is called as Non-parametric function or function without parameters.

> Syntax of non-parametric function:
>
> ```
> def function_name():
>     // body of function
>     return value
> ```

Ex 7.5 (Non-parametric Function)

```
# defining a function
def display():
    print("Welcome to facebook.")

# calling function
display()
```

```
Welcome to facebook.
```

Lesson Two : Function with Argument

What is Parametric function?

A function which has parameter(s) in the function definition as an input is called as parametric function or function with argument.

What is Argument?

An argument is a variable (which contains data) or a parameter which is sent to the function as input.

Before getting into argument types, let's get familiar with words formal and actual arguments.

1. Formal arguments

When a function is defined it (may) has (have) some parameters within the parentheses. These parameters, which receive the values sent from the function call, are called formal arguments.

2. Actual arguments

The parameters which we use in the function call or the parameters which we use to send the values/data during the function calling are called actual arguments.

In the example down below:

- a and b are formal arguments.
- 7 and 3 are actual arguments.

Ex 7.6 (Parametric Function)

```
# defining a function
def add(a,b):
    c = a+b
    print(c)

# calling a function
add(7, 3)
```

```
10
```

EXERCISE 7.4

Create a parametric function to check a number is even or odd?

```
# write your answer code here

```

Types of Arguments

In python, depending on the way or format we send the arguments to the function, the arguments can be classified into five types:

1. Positional arguments
2. Keyword arguments
3. Default arguments
4. Variable length arguments
5. keyword variable length argument

1. Positional Arguments

If there are three arguments (formal arguments) in the function definition, then you need to send three arguments(actual arguments) while calling the function. The actual arguments will be received by the formal arguments in the same order as in the function calling.

Ex 7.7 (Positional Arguments)

```
# defining a function
def sub(a,b):
    c = a - b
    print(c)

# calling a function
sub(5, 2)
```

```
3
```

Let's understand this through an example. In the above example the actual arguments 5, 2 will be received in the same order sent i.e. 5 will go into variable 'a', and 2 will go into variable 'b'.

The number of arguments and position of arguments should be matched, otherwise we will get errors as shown below.

Ex 7.8 (Positional arguments Error)

```
# defining a function
def sub(a, b):
    c = a - b
    print(c)

# calling function
sub(10, 20, 30)
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
Cell In[8], line 7
      4     print(c)
      6 # calling function
----> 7 sub(10, 20, 30)

TypeError: sub() takes 2 positional arguments but 3
```

If we send the arguments as 2, 5 then 2 will go into 'a' and 5 will go into 'b' then we will get different result(output).

Ex 7.9 (Positional arguments)

```
# defining a function
def sub(a, b):
    c = a - b
    print(c)

# calling function
sub(2, 5)
```

```
-3
```

💡 *How to resolve the above problem?*

2. Keyword Arguments

In the function call, if we send the values using keys then they are called keyword arguments. Here, keys are nothing but the names of the variables. In a way, we can say that keyword arguments are arguments that recognize the parameters by the name of the parameters.

Ex 7.10 (Keyword arguments)

```
# defining a function
def sub(a, b):
    c = a - b
    print(c)

# calling a function
sub(b=2, a=5)
```

```
3
```

At the time of calling this function, we must pass two values and we can write which value is for what by using name of the parameter.

> Syntax:
> `function_name(keyword = actual_argument)`

PS. a and b are called as keywords in the above scenario. Here we can change the order of arguments.

`EXERCISE 7.5`

`Create a function to print student id and student name using keyword arguments?`

```
# write your answer code here

```

💡 *Can we use both positional and keyword arguments? Yes of course…*

3. Positional and keyword arguments

We can use both positional and keyword arguments simultaneously. But first we must take positional arguments and then keyword arguments, otherwise we will get syntax error.

Ex 7.11 (Positional and keyword arguments)

```
# defining a function
def sub(a, b):
    c = a - b
    print(c)

# calling a function
sub(5, b=2)
```

```
3
```

Ex 7.12 (Positional and keyword arguments Error)

```
# defining a function
def sub(a, b):
    c = a - b
    print(c)

# calling a function
sub(a=5, 2)
```

```
  Cell In[12], line 7
    sub(a=5, 2)
               ^
SyntaxError: positional argument follows keyword
argument
```

EXERCISE 7.6

Create a function to print phone brand and phone price using positional keyword arguments?

```
# write your answer code here

```

If you don’t pass exact number of parameters you will get a Type error.

Ex 7.13 (Positional arguments TypeError)

```
# defining a function
def sub(a, b):
    c = a - b
    print(c)

# calling function
sub(a=5)
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
Cell In[13], line 7
      4     print(c)
      6 # calling function
----> 7 sub(a=5)

TypeError: sub() missing 1 required positional
```

💡 *How to resolve the above error?*

4. Default Arguments

In the function definition, while declaring parameters, we can assign some value to the parameters, which are called default values. Such default values will be considered when the function call does not send any data to the parameter.

Let's take earlier function as an example. There, the function sub() has parameters as a and b. To resolve TypeError, we can set the default value of b as 0, then default values will be overridden with passed value.

Ex 7.14 (Default arguments)

```
# defining a function
def sub(a, b=0):
    c = a - b
    print(c)

# calling a function
sub(a=5, b=2)

# calling a function
sub(a=5)
```

```
3
5
```

Note: If we are not passing any value, then only default value will be considered. While defining a function, after default arguments we should not take non-default arguments.

Ex 7.15 (Default arguments Error)

```
# defining a function
def add(a=3, b):
    print(a + b)

# calling a function
add(a=5, b=2)

# calling a function
add(a=9)
```

```
  Cell In[16], line 2
    def add(a=3, b):
                  ^
SyntaxError: non-default argument follows default
argument
```

EXERCISE 7.7

Create a function to print football player name and no. of goals he scored using default arguments?

Default no. of goal is 0

```
# write your answer code here

```

💡 *What if you don't know number of values to pass?*

5. Variable Length Arguments

Sometimes, the programmer (you) does not know how many values need to pass to function. In that case, the programmer cannot decide how many arguments to be given in the function definition. Therefore, we use variable length arguments to accept n number of arguments.

The variable length argument is an argument that can accept any number of values.

The variable length argument is written with a '*' (one star) before variable in function definition.

> Syntax for Variable length Arguments:
> `def function_name(x, *y):`
> `    //body of function`
> `    return value`

x is formal argument, *y is variable length argument. Now we can pass any number of values to this *y.

PS. Internally, provided values will be represented in tuple.

Ex 7.16 (Variable length argument)

```
# defining a function
def detail(name, *others):
    print(name, others)

# calling a function
detail("John")

# calling a function
detail("John", 20)

# calling a function
detail("John", 20, 3.7)
```

```
John ()
John (20,)
John (20, 3.7)
```

EXERCISE 7.8

Create a function to print player name and other details of him using variable length arguments?

```
# write your answer code here

```

💡 *How to identify other values?*

keyword variable length argument (variable)**

Just as variable length arguments, there are keyword variable length arguments which are n key value pairs.

> The syntax is given below:
> `def function_name(**x):`
> `    // body of function`
> `    return value`

**x represents as keyword variable argument. Internally it represents like a dictionary. A dictionary stores the data in the form of key value pairs.

Ex 7.17 (keyword variable length argument)

```
# defining a function
def detail(**others):
    print(others)

# calling a function
detail(name='John', age=20, gpa=3.7)
```

```
{'name': 'John', 'age': 20, 'gpa': 3.7}
```

Lesson Three : Function as a Parameter

Function as a variable

PS. We can also assign return value to a variable.

> Syntax to use Assign return value to a variable:
>
> `def function_name():`
>
> `        // body of function`
>
> `        return value`
>
> `variable_name = function_name()`

Ex 7.18 (Function returning the value as variable)

```
# defining a function
def add(a, b):
    return a + b

# calling a function
x = add(1, 2)

print("Sum of two numbers is :",x)
```

```
Sum of two numbers is : 3
```

`EXERCISE 7.9`

`Create a function to calculate average of two numbers and assign return value to a variable?`

```
# write your answer code here

```

Function calling another function

It is also possible in python that a function can call another function.

> Syntax to call a function from another function:
>
> ```
> def function_one():
>     // body of function one
>     return value
>
> def function_two():
>     // body of function two
>     function_one() # This is optional
>     return value
>
> function_two()
> ```

Ex 7.19 (Function calling another function)

```
1 # defining a function
2 def add(a,b):
3     return(a+b)
4
5 # defining a function
6 def average(a,b):
7     sum = add(a,b)
8     avg = sum / 2
9     print("Average =",avg)
10
11 # calling a function
12 average(7,3)
```

```
Average = 5.0
```

EXERCISE 7.10

Create a function to calculate tan(45°) using two other functions sine and cosine return values by using eval() function?

```
1 # write your answer code here
2
```

Nested Function

We can define one function inside another function. If we defined the inner function, then we need to call that inner function in the outer function.

> Syntax for nested function:
>
> ```
> def function_one():
>     // body of function one
>
>     def function_two():
>         // body of function two
>         return of function two
>
>     function_two()
>     return of function one
> function_one()
> ```

In the example down below:

- First, we define a function `average()`
- Then we define `add()` function inside `average()` function to calculate sum of two numbers.
- Finally we calculate average based on return value of `add()` function.

Ex 7.20 (Nested function)

```
1 # defining a function 1
2 def average(a, b):
3     # defining a function 2
4     def add():
5         return a + b
6 
7     # calling a function 1
8     sum = add()
9 
10    avg = sum / 2
11    print("Average =",avg)
12 
13 # calling a function 2
14 average(7, 3)
```

```
Average = 5.0
```

EXERCISE 7.11

Create a function to calculate tan(45°) using two other nested functions sine and cosine eval() functions?

```
# write your answer code here

```

Function as a parameter

PS. We can pass function as a parameter to another function.

> Syntax to pass a function as a parameter:
> `function_two(function_one())`

Ex 7.21 (pass a function as parameter)

```
# defining a function 1
def add(a, b):
    return a + b

# defining a function 2
def average(sum):
    avg = sum / 2
    print("Average =",avg)

# calling a function 2
average(add(7, 3))
```

```
Average = 5.0
```

EXERCISE 7.12

Create a function to calculate tan(45°) by passing sin & cos functions as parameter eval() functions?

```
# write your answer code here

```

💡 *Can we call a function inside its definition block?*

What is Recursive Function?

A function is called recursive when it is called by itself. Let's understand this with an example.

Let's consider a function which calculates the factorial of a number. It can be written as a recursive functions as explained below.

3! = 3 x 2 x 1

factorial(3) = 3 x factorial(2)

factorial(2) = 3 x 2 x factorial(1)

factorial(1) = 3 x 2 x 1 x factorial(0)

factorial(n) = n x factorial(n-1)

Ex 7.22 (Factorial using recursive function)

```
# defining a function
def factorial(n):
    if n == 0:
        result = 1
    else:
        result = n * factorial(n-1)
    return result

# calling a function
x = factorial(4)

print("Factorial of 4 is :",x)
```

```
Factorial of 4 is : 24
```

PS. We will see Factorial without recursive function later in control Flow /Loops/ Chapter

EXERCISE 7.13

Create a function to calculate sum of all numbers from 1 to 100?

```
# write your answer code here

```

Advantages of using recursive functions:

- We can reduce length of the code and improves readability.
- We can solve complex problems in a very easy way.

Lesson Four : Types of Variables

As we discussed in variables chapter about properties of variables, The variables based on their scope can be classified in to two types:

- Local variables
- Global variables

1. Local Variables

The variables which are declared inside of the function are called as local variables. Their scope is limited to the function i.e. we can access local variables within the function only.

Ex 7.23 (Local variables)

```
# defining a function
def add():
    a, b = 5, 2
    print(a + b)

# calling a function
add()
```

```
7
```

If we are trying to access local variables outside of the function, then we will get an error.

Ex 7.24 (Local variables Error)

```
# defining a function
def add():
    a, b = 5, 2
    print(a + b)

# defining a function
def sub():
    print(a - b)

# calling a function
sub()
```

```
---------------------------------------------------------
--------------------
NameError                                 Traceback
(most recent call last)
Cell In[33], line 11
      8     print(a - b)
     10 # calling a function
---> 11 sub()

Cell In[33], line 8, in sub()
      7 def sub():
----> 8     print(a - b)

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

2. Global Variables

The variables which are declared outside of the function are called global variables. Global variables can be accessed in all functions.

Ex 7.25 (Global variables)

```
a, b = 5, 2

# defining a function
def add():
    print(a + b)

# defining a function
def sub():
    print(a - b)

# calling a functions
add()
sub()
```

```
7
3
```

💡 *What if both local and global variables have the same name?*

When can we choose a global keyword?

There might be some scenarios, where the global variable names and local variable names are the same. In such cases, within the function, by default, the local variables are only referred and the global variables are ignored.

Ex 7.26 (same global and local variables name)

```
a, b = 5, 2

# defining a function
def add():
    a = 9
    print(a + b)

# defining a function
def sub():
    print(a - b)

# calling a function
add()
sub()
```

```
11
3
```

If we want our function to refer to global variable rather than the local variable, we can use global keyword before the variable in the function as shown in the example below.

Ex 7.27 (same global and local variables name)

```
a, b = 5, 2

# defining a function
def add():
    global a
    a = 9
    print(a + b)

# defining a function
def sub():
    print(a - b)

# calling a functions
add()
sub()
```

```
11
7
```

PS. The keyword "global" can be used for the following two purposes:

1. To declare global variable inside function.
2. To make global variables available to the function.

If we use the global keyword inside the function, then the function is able to read only global variables.

PROBLEM: This would make the local variable no more available.

💡 *How to solve this problem?*

globals() built-in function

The problem of local variables not available, due to use of global keyword can be overcome by using Python built-in function called `globals()`.

`globals()` is a built in function which returns a table of current global variables in the form of a dictionary. Using this function, we can refer to the global variable as:

> Syntax:
> `global()["variable_name"]`

Ex 7.28 (globals built-in function)

```
a, b = 5, 2

# defining a function
def add():
    a = 9
    print(a + b)
    print(globals()['a'] + b)

# calling a function
add()
```

```
11
7
```

Lesson Five : Lambda (Anonymous) Function

The name of the function is the mandatory item in the function definition as discussed earlier. But, in python, we have a keyword called ‘lambda’ with which we can define a simple function in a single line without actually naming it. Such functions are called lambda functions.

> Syntax of lambda function:
> `lambda arguments_list : expression`

Ex 7.29 (Lambda Function)

```
s = lambda a: a * a
x = s(4)
print(x)
```

```
16
```

Ex 7.30 (Lambda Function)

```
# defining a function
s = lambda a, b : a * b

# calling a function
x = s(4, 5)

print(x)
```

```
20
```

Points to Remember while working with Python Lambda Functions:

- A lambda function can take any number of arguments but should have only one expression.
- It takes the parameters and does some operation on them and returns the result, just the same as normal functions.
- The biggest advantage of lambda functions is that a very concise code can be written which improves the readability of the code.
- This can be used for only simple functions but not for complex functions.
- The lambda function, mostly, can be used in combination with other functions such as map(), reduce(), filter() etc. which we are going to discuss now.

`EXERCISE 7.14`

`Create a lambda function to calculate average of two numbers?`

```
# write your answer code here

```

1. filter() Function

This function is used to filter values from a sequence of values.

> Syntax for filter() function:
> `filter(function_name, sequence)`

The filter function will filter the elements in the sequence based on the condition in the function. Let's understand it through the example.

Ex 7.31 (filter() function)

```
a = [1, 2, 3, 4]
g = filter(lambda x : x > 2, a)
x = list(g)
print(x)
```

```
[3, 4]
```

In the above example, filter applies the lambda function on all the elements in the ‘a’ and returns the elements which satisfy the lambda function.

`EXERCISE 7.15`

`Create a filter function to check a number is even or odd?`

```
# write your answer code here

```

2. map() Function

This function is used to map a particular function onto the sequence of elements. After applying, this returns a new set of values.

> Syntax of map() function:
> `map(function_name, sequence)`

Ex 7.32 (map() function)

```
a = [1, 2, 3, 4]
g = map(lambda x : x+2, a)
x = list(g)
print(x)
```

```
[3, 4, 5, 6]
```

`EXERCISE 7.16`

`Create a list variable and store price of foods then print all foods price after 25% discount using map() function?`

```
# write your answer code here

```

3. reduce() Function

This function reduces the sequence of elements into a single element by applying a specific condition or logic. To use the reduce function we need to import the functools module.

> Syntax for reduce() function:
> `from functools import reduce`
> `reduce(function_name, sequence)`

Ex 7.33 (reduce() function)

```
from functools import reduce
a = [1, 2, 3, 4]
g = reduce(lambda x, y : x+y, a)
print(g)
```

```
10
```

EXERCISE 7.17

Create a list variable and store your mid semester Exam results of each subject then find its average using reduce() function?

```
# write your answer code here

```

Lesson Six : Decorators and Generators

What is Decorator?

A decorator is a special function which adds some extra functionality to an existing function.

A decorator is a function that accepts a function as a parameter and returns a function.

Decorators are useful to perform some additional processing required by a function.

Steps to create decorator:

- Step 1: Decorator takes a function as an argument.
- Step 2: Decorator body should have inner function.
- Step 3: Decorator should return a function.
- Step 4: The extra functionality which you want to add to a function can be added in the body of the inner_function.

> Syntax
>
> ```
> def decor(func):
>     def inner_function():
>         // body of inner function
>         return inner_function
> ```

PS. Here func is the argument/parameter which receives the function.

💡 *Suppose you want to add some extra functionality of adding the two numbers only if they are positive. If any number is negative, then take it as 0 during adding. For adding this extra functionality let us create a decorator.*

> Syntax:
>
> ```
> def inner_function(x, y):
>     if x < 0:
>         x = 0
>     if y < 0:
>         y = 0
>     return func(x, y)
> return inner_function
> ```

We have created our decorator and now let's use it with our add function from

```
add = decor(add)
```

With the above statement, we are passing the add function as parameter to the decorator function, which is returning inner_function.

After this, whenever we call add, the execution goes to inner_function in the decorator.

```
add(-10, 20)
```

In inner_function, we are doing the extra logic for checking whether the arguments are positive or not. If not positive we are assigning them with zero. And we are passing the processed values to the original add function which was sent to the decorator.

Our final code will be as follows

Ex 7.34 (Decorator Function)

```
# defining a function
def decor(func):

    # defining inner function
    def inner_function(x, y):
        if x < 0:
            x = 0
        if y < 0:
            y = 0
        return func(x, y)

    return inner_function

# defining add function
def add(a, b):
    res = a + b
    return res

# calling a function
add = decor(add)

print(add(20, 30))
print(add(-10, 5))
```

```
50
5
```

EXERCISE 7.18

Create a decorator to divide two numbers but if a number is divided by zero, print "Invalid"?

```
# write your answer code here

```

@ symbol in python

In the above example, in order to use the decorator, we have used the **'add = decor(add)'** line. Rather than using this we can just use the **'@decor'** symbol on top of the function for which we want to add this extra functionality. The decorator once created can also be used for other functions as well.

Ex 7.35 (Subtract Function using decorator)

```
# defining a function
def decor(func):

    # defining a function
    def inner_function(x, y):
        if x < 0:
            x = 0
        if y < 0:
            y = 0
        return func(x, y)

    return inner_function

@decor
# defining a function
def sub(a, b):
    res = a - b
    return res

# calling a function
print(sub(30, 20))
print(sub(10, -5))
```

```
10
10
```

EXERCISE 7.19

Create a decorator to divide two numbers but the numerator is always greater than denominator?

```
# write your answer code here

```

What are Generators?

Generators are just like functions which give us a sequence of values one as an iterable. Generators contain yield statements just as functions contain return statements.

next() function in Python:

If we want to retrieve elements from a generator, we can use the next function on the iterator returned by the generator. This is the other way of getting the elements from the generator.

Advantages of using generators

- **Memory Efficiency:** Generators save memory by generating values on-the-fly instead of storing them all at once.
- **Lazy Evaluation:** Values are computed only as needed, improving efficiency for expensive operations.
- **Iteration Support:** Generators are iterable, making it easy to loop over generated values.
- **Simplified Code:** Generators help break complex code into smaller, more manageable pieces.
- **Infinite Sequences:** Generators handle large or infinite sequences that can't fit in memory.

Ex 7.36 (Generators)

```
# defining a function
def my_gen():
    n = 1
    print(n)
    yield n

    n += 1
    print(n)
    yield n

    n += 1
    print(n)
    yield n

# calling a function
a = my_gen()

print(type(a))
next(a)
next(a)
next(a)
```

```
<class 'generator'>
1
2
3
3
```

`yield` is a keyword used in the context of generators. Generators are special functions that can be paused and resumed during their execution. When a function contains the `yield` keyword, it becomes a generator function.

Think of a generator function as a recipe for creating a sequence of values. Instead of generating all the values at once and storing them in memory, the generator function generates one value at a time and "yields" it back to the caller. The generator function remembers its state, so when it's called again, it continues from where it left off.

Now, let's talk about the `next()` function. The `next()` function is used to manually iterate over the values produced by a generator. When you call `next()` on a generator object, it resumes the execution of the generator function and returns the next value produced by the `yield` statement.

`EXERCISE 7.20`

`Create a generator to print top 3 athletes who finish Olympic marathon 2019?`

```
# write your answer code here

```

Key Takeaways

- *A function is a group of statements or a block of code to perform a certain task.*
- *If a group of statements is repeatedly required, then it is not recommended to write a function.*
- *There are two types of functions: Pre-defined (built-in) and User-defined.*
- *An argument is a variable (which contains data) or a parameter which is sent to the function as input.*
- *Nested function is a function that is defined inside another function.*
- *A function is called recursive when it is called by itself.*
- *Variables can be classified into two types based on their scope: Local variables and Global variables.*
- *Lambda function is a short anonymous function that can be defined inline without a name.*
- *Decorator is a special function which adds some extra functionality to an existing function.*
- *Generators are just like functions which give us a sequence of values one as an iterable.*

// END OF CHAPTER SEVEN

Click here for Chapter Eight

CHAPTER EIGHT

String

Objectives

- *What is a string?*
- *How can you access a string?*
- *What is indexing?*
- *What is slicing?*
- *What operations can you perform on a string?*

Lesson One : Introduction to String

We already learn the first Hello World program in python. In that program we just print a group of characters by using print() function. Those groups of characters are called as a string.

> Example: Python Program to print string
>
> ```
> print("Hello World")
> ```
>
> ```
> Hello World
> ```

What is a string?

A group of characters enclosed within single or double or triple quotes is called as string. We can say string is a sequential collection of characters.

How to create a String?

There are four ways to create a string.

> Syntaxs :
>
> `string_name = 'String' #single quotes`
>
> `string_name = "String" #double quotes`
>
> `string_name = '''String''' #triple single quotes`
>
> `string_name = """String""" #triple double quotes`

Note: Generally, to create a string mostly used syntax is double quotes syntax.

When triple single and triple double quotes are used?

If you want to create multiple lines of string, then triple single or triple double quotes are the best to use.

Ex 8.1 (Creating a string using four ways)

```
name = 'Michael Jackson'
music = "Michael’s song"
s = '"Michael’s song is awesome."'
print("My name is ",name)
print("This is ",music)
print("{} Thomas D.".format(s))
```

```
My name is  Michael Jackson
This is  Michael’s song
"Michael’s song is awesome." Thomas D.
```

How to create an empty string?

If a string has no characters in it then it is called an empty string.

Ex 8.2 (Creating an empty string)

```
s = ""
print("This is empty string ",s)
```

```
This is empty string
```

Lesson Two : Accessing a String

How to access a String?

There are three ways in which you can access the characters in the string. They are:

- By using indexing
- By using slicing operator
- By using loops (*PS. we will discuss in later Control Flow Chapter*)

What is Indexing?

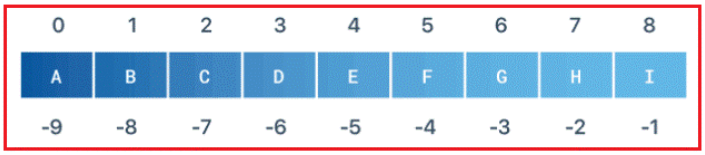
Indexing means a position of string's characters where it stores. We need to use square brackets [ ] to access the string index. String indexing result is string type. String indices should be integer otherwise we will get an error. We can access the index within the index range otherwise we will get an error.

Python supports two types of indexing

- **Positive indexing:** The position of string characters can be a positive index from left to right direction (we can say forward direction). In this way, the starting position is 0 (zero).

- **Negative indexing:** The position of string characters can be negative indexes from right to left direction (we can say backward direction). In this way, the starting position is -1 (minus one).

Diagram representation



Note: If we are trying to access characters of a string with out of range index, then we will get error as IndexError.

Ex 8.3 (Accessing a string with index)

```
1 name = "Michael Jackson"
2 print(name[0])
3 print(name[-7])
```

```
M
J
```

Ex 8.4 (Accessing a string with float index)

```
1 name = "Michael Jackson"
2 print(name[1.3])
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
Cell In[8], line 2
      1 name = "Michael Jackson"
----> 2 print(name[1.3])

TypeError: string indices must be integers, not 'float'
```

Ex 8.5 (string with out of bound index)

```
1 name = "Michael Jackson"
2 print(name[100])
```

```
---------------------------------------------------------
--------------------
IndexError                                Traceback
(most recent call last)
Cell In[9], line 2
      1 name = "Michael Jackson"
----> 2 print(name[100])

IndexError: string index out of range
```

EXERCISE 8.1

Create a string variable "name" and store your name then print only vowel letters from your name?

```
1 # write your answer code here
2
```

💡 *What if we want to access a word from a string?*

In such scenarios we use string slicing.

What is Slicing?

A substring of a string is called a slice. A slice can represent a part of string from string or a piece of string. String slicing result is string type. We need to use square brackets [ ] in slicing.

In slicing we will not get any Index out of range exception. In slicing indices should be integer or None otherwise we will get errors.

> Syntax:
> `string_name[start : stop : step]`

- **start:** Represent from the starting index position (default is 0)
- **stop:** Represents the (ending index-1) position (default is last index)
- **step:** Represents the increment the position of the index while accessing (default is 1)

Note: If you are not specifying the begin index, then it will consider the beginning of the string. If you are not specifying the end index, then it will consider the end of the string. The default value for step is 1.

Ex 8.6 (String Slicing)

```
text = "together"
print(text [:])
print(text[:2])
print(text[2:5])
print(text[5:])
print(text[::3])
print(text[1:8:6])
```

```
together
to
get
her
tee
or
```

EXERCISE 8.2

Create a string variable "quote" and store your favorite quote then print each words from the quote?

```
# write your answer code here

```

What is String Immutable?

Once we create a variable then the state of the existing variable cannot be changed or modified. This behavior is called immutability.

PS. Strings are immutable in Python.

String having immutable nature. Once we create a string variable then we cannot change or modify it.

Ex 8.7 (String Immutability)

```
name = "Michael"
name[0] = "P"
print(name)
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
Cell In[11], line 2
      1 name = "Michael"
----> 2 name[0] = "P"
      3 print(name)

TypeError: 'str' object does not support item
```

Lesson Three : String Operations (using string operators)

💡 *What if we want to merge two strings?*

1. Concatination (+)

The + operator works like concatenation or joins the strings. While using + operator on string, It is compulsory that both variables should be string type, otherwise we will get an error.

> Syntax:
> `s1 + s2`

Ex 8.8 (Concatenation + Operator)

```
1 fn = "Michael "
2 ln = "Jackson"
3 print(fn + ln)
```

```
Michael Jackson
```

Ex 8.9 (Concatenation Error)

```
1 fn = "Michael"
2 num = 4
3 print(fn + num)
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
Cell In[13], line 3
      1 fn = "Michael"
      2 num = 4
----> 3 print(fn + num)

TypeError: can only concatenate str (not "int") to str
```

`EXERCISE 8.3`

`Create four string variables "UPPER", "lower", "Symbols", "digits" and store uppercase letters, lowercase letters, special symbols and digits(0-9) respectively then create a password which contains all of them?`

```
1 # write your answer code here
2
```

💡 *What if we want to duplicate a string?*

2. Multiplication (*)

This is used for string repetition. While using * operator on string, It is compulsory that one variable should be string and other arguments should be int type.

> Syntax:
> `s * number`

Ex 8.10 (Multiplication * Operator)

```
fn = "Michael "
rn = 4
print(fn * rn)
```

```
Michael Michael Michael Michael
```

Ex 8.11 (Multiplication Error)

```
fn = "Michael"
rn = "4"
print(fn * rn)
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
Cell In[15], line 3
      1 fn = "Michael"
      2 rn = "4"
----> 3 print(fn * rn)

TypeError: can't multiply sequence by non-int of type
```

Lesson Four : String Operations (using string methods)

💡 *What if we want to know no. of characters in a string?*

1. len() method

We can find the length of string by using len() function. By using the len() function we can find groups of characters present in a string. The len() function returns int as result.

> Syntax:
> `len(s)`

Ex 8.12 (len() Function)

```
fn = "Michael"
print("Length of string:",len(fn))
```

```
Length of string: 7
```

EXERCISE 8.4

Create a string variable "quote" and store your favorite quote then find length of the quote?

```
# write your answer code here

```

💡 *What if we want to check whether a word / letter / is found in a string?*

2. Membership operators

We can check, if a string or character is a member/substring of string or not by using below operators:

- in
- not in

in operator returns True, if the string or character found in the main string.

Ex 8.13 (in operator)

```
fn = "Michael"
print('M' in fn)
print('m' in fn)
print('Mic' in fn)
print('lm' in fn)
```

```
True
False
True
False
```

not in operator returns the opposite result of in operator. not in operator returns True, if the string or character is not found in the main string.

Ex 8.14 (not in operator)

```
S = "pair"
print('air' not in S)
print('chair' not in S)
```

```
False
True
```

EXERCISE 8.5

Create a string variable "name" and store a string “Yonatan” then remove all vowel letters from the string?

```
# write your answer code here

```

3. Relational Operators

We can use the relational operators like >, >=, <, <=, ==, != to compare two strings. While comparing it returns boolean values, either True or False. Comparison will be done based on alphabetical order or dictionary order.

Ex 8.15 (Comparison operators on string) / User name validation /

```
un = "michael"
i = str(input("Enter user name:"))

if i == un:
    print("Hello", i)
    print("Welcome to facebook.")
else:
    print("Invalid user name!")
```

```
Enter user name:michael
Hello michael
Welcome to facebook.
```

💡 *A space is also considered as a character inside a string. Sometimes unnecessary spaces in a string will lead to wrong results.*

Ex 8.16 (White space string Error)

```
un = "michael"
i = str(input("Enter user name:"))

if i == un:
    print("Hello", i)
    print("Welcome to facebook.")
else:
    print("Invalid user name!")
```

```
Enter user name:mic hael
Invalid user name!
```

4. strip()

Predefined methods to remove spaces in Python

- rstrip() - remove spaces at right side of string
- lstrip() - remove spaces at left side of string
- strip() - remove spaces at left & right sides of string

Note: These methods won't remove the spaces which are in the middle of the string.

> Syntax:
> `s.strip()`

Ex 8.17 (Remove Space from a String)

```
1 i = "to get her"
2 i = i.strip()
3 print(i)
```

```
to get her
```

`EXERCISE 8.6`

`Create username validation program excluding whitespace?`

```
1 # write your answer code here
2
```

Searching Methods

We can find a substring in two directions like forward and backward direction. We can use the following 4 methods:

For forward direction

- find()
- index()

For backward direction

- rfind()
- rindex()

5. find() method

This method returns an index of the first occurrence of the given substring.

If it is not available, then we will get -1 (minus one) value. By default find() method can search the total string of the object.

> Syntax:
> `string_name.find(substring)`

You can also specify the boundaries while searching.

> Syntax:
> `string_name.find(substring, begin, end)`

PS. It will always search from begin index to end - 1 index.

Ex 8.18 (finding substring by using find method)

```
fn = input("Enter your name: ")
sub = input("Enter nick name: ")
i = fn.find(sub)

if i == -1:
    print(sub, "is not found in",fn)
else:
    print(sub, "is found in",fn)
```

```
Enter your name: Michael
Enter nick name: Mic
Mic is found in Michael
```

EXERCISE 8.7

Create a string variable "quote" and store your favorite quote then find letter "x" in the quote?

```
# write your answer code here

```

6. index() method

index() method is exactly the same as find() method except that if the specified substring is not available then we will get ValueError. So, we can handle this error in application level.

> Syntax:
> `string_name.index(substring)`

Ex 8.19 (finding substring by using index method)

```
s = "Together"
print(s.index("get"))
```

```
2
```

Ex 8.20 (finding substring by using index method)

```
s = "Together"
print(s.index("to"))
```

```
---------------------------------------------------------
--------------------
ValueError                                Traceback
(most recent call last)
Cell In[25], line 2
      1 s = "Together"
----> 2 print(s.index("to"))

ValueError: substring not found
```

`EXERCISE 8.8`

`Create a string variable "quote" and store your favorite quote then find index of letter "a" in the quote?`

```
# write your answer code here

```

💡 *What if we want to find total no. of a specific letter or word in a string?*

7. count() method

By using count() method we can find the number of occurrences of substring present in the string

> Syntax:
> `string_name.count(substring)`

Ex 8.21 (count() method)

```
fn = "Michael Jackson"
print(fn.count("a"))
print(fn.count("j"))
```

```
2
0
```

`EXERCISE 8.9`

`Create a string variable "name" and store your name then find no. of vowel letters in your name?`

```
# write your answer code here

```

💡 *What if we want to replace a letter or word?*

8. replace() method

We can replace old string with new string by using replace() method. As we know string is immutable, so replace() method never performs changes on the existing string object. The replace() method creates a new string object, we can check this by using id() method.

> Syntax:
> `string_name.replace(old string, new string)`

Ex 8.22 (replace a string with another string)

```
s1 = "Java programming language"
s2 = s1.replace("Java", "Python")
print(s1)
print(s2)
```

```
Java programming language
Python programming language
```

Does replace() method modify the string objects?

A big NO. String is immutable. Once we create a string object, we cannot change or modify the existing string object. This behavior is called immutability.

If you are trying to change or modify the existing string object, then with those changes a new string object will be created. So, replace() method will create a new string object with the modifications.

EXERCISE 8.10

Create a string variable "email" and store your gmail then convert it into yahoo mail?

```
# write your answer code here

```

💡 *What if we want to split a string?*

9. split() method

We can split the given string according to the specified separator by using split() method. The default separator is space. split() method returns list.

> Syntax:
> `new_string = string_name.split(separator)`

Ex 8.23 (Splitting a string)

```
s = "to get her"
split_s = s.split()
print("Before split:",s)
print("After split:",split_s)
print(type(split_s))
```

```
Before split: to get her
After split: ['to', 'get', 'her']
<class 'list'>
```

Ex 8.24 (Splitting a string based on separator)

```
text = "Michael, He was good Artist."
n = text.split(",")
print(n)
print(type(n))
```

```
['Michael', ' He was good Artist.']
<class 'list'>
```

EXERCISE 8.11

Create a string variable "name" and store your full name then split it?

```
# write your answer code here

```

💡 *What if we want to join the two splitted strings?*

10. join() method

We can join a group of strings (list or tuple) with respect to the given separator.

> Syntax:
> `string_name = separator.join(A)`

PS. variable "A" can be any sequence data type.

Ex 8.25 (joining a string)

```
1 split_s = ['to','get','her']
2 s = '-'.join(split_s)
3 print(split_s)
4 print(s)
```

```
['to', 'get', 'her']
to-get-her
```

`EXERCISE 8.12`

`Create a list variable "name" and store your first name, your middle name and surname then join it?`

```
1 # write your answer code here
2
```

💡 *What if we want to change cases of a string?*

Changing cases of string

- **upper()** – This method converts all characters into upper case
- **lower()** – This method converts all characters into lower case
- **swapcase()** – This method converts all lower-case characters to uppercase and all upper-case characters to lowercase
- **title()** – This method converts all character to title case (First character in every word will be in upper case and all remaining characters will be in lower case)
- **capitalize()** – Only the first character will be converted to upper case and all remaining characters can be converted to lowercase.

Ex 8.26 (Changing cases)

```
fn = 'MicHaEl jACksON'
print(fn.upper())
print(fn.lower())
print(fn.swapcase())
print(fn.title())
print(fn.capitalize())
```

```
MICHAEL JACKSON
michael jackson
mIChAeL JacKSon
Michael Jackson
Michael jackson
```

Password Validation

Ex 8.27 (Password Validation)

```
p = str(input("Enter Password: "))

if len(p) > 8:
    if p.islower()==False:
        if p.isupper()==False:
            if p.isdigit()==False:
                print('Password set Successfully')
            else:
                print('No Digit')
                del p
        else:
            print('No Uppercase')
            del p
    else:
        print('No Lowercase')
        del p
else:
    print('Less Characters')
```

```
Enter Password: 123AVBsaf#$
Password set Successfully
```

`EXERCISE 8.13`

`Create a Instagram username validation program using the following rules?`

- `Minimum 5 characters`
- `Maximum 10 characters`
- `Only lowercase letters`
- `Underscore and dot is allowed`
- `Whitespace, symbol are not allowed`

```
# write your answer code here

```

Finally What about Character data type?

If a single character stands alone then we can say that is a single character. For example: M for Male F for Female.

In other programming languages char data type represents the character type, but in python there is no specific data type to represent characters. We can fulfill this requirement by taking a single character in single quotes. A single character also represents as string type in python.

Ex 8.28 (Character data type)

```
gender1 = "M"
gender2 = "F"
print(gender1)
print(gender2)
print(type(gender1))
print(type(gender2))
```

```
M
F
<class 'str'>
<class 'str'>
```

Key Takeaways

- *String is a sequence of characters.*
- *You can access elements of a string using indexing, slicing, or loops.*
- *Indexing refers to the position of characters within a string.*
- *Slicing refers to extracting a substring from a string.*
- *You can use operators (+, *, membership and relational operators) and methods { `strip()`, `find()`, `index()`, `count()`, `replace()`, `split()`, `join()` } on a string.*
- *You can change cases of a string using `upper()`, `lower()`, `swapcase()`, `title()`, `capitalize()`.*

// END OF CHAPTER EIGHT

Click here for Chapter Nine

CHAPTER NINE

List and Tuple

Objectives

- *What is data structure?*
- *What is List?*
- *How can you access elements of a list?*
- *How to insert elements in a list?*
- *How to delete elements in a list or the list itself?*
- *How to arrange elements of a list?*
- *What is nested list?*
- *What is list aliasing and cloning?*
- *What operations can you perform on a list?*
- *What is tuple?*
- *What is the difference between List and Tuple?*

Lesson One : Introduction to List

While we write a program, we store a lots of data, then those data become cluttered. Finally it will be much difficult to retrieve, access and use them. Therefore we need to arrange, organize and manage those data by grouping /categorizing/ then store in a single variable. In programming, this process is called as Data Structure.

PS. Data Structure is a particular way of organizing data in a computer so that it can be used effectively.

What is List?

A list is a data structure which can store a group of elements or objects in a single variable.

Properties of List

- Lists can store the same type as well as different types of elements. Hence, it is **heterogeneous**.
- Lists have **dynamic** nature which means we don't need to mention the size (number of elements) of the list while declaring. The size will increase dynamically.
- **Insertion order is preserved.** The order, in which the elements are added into a list, is the order in which output is displayed.
- List can store **duplicates** elements.
- In lists **index** plays the main role. Using index we perform almost all the operations on the elements in it.
- List objects are **mutable** which means we can modify or change the content of the list even after the creation

How to create a List?

There are a couple of ways to create a list in Python. Let's see them with examples.

- Using square brackets
- Using list() function

1. Creating a list using square brackets '[ ]'

We can create a list by using square brackets '[]'. The elements in the list should be comma separated.

> Syntax :
> `list_name = [elements]`

Ex 9.1 (Creating an empty list)

```
1 l = []
2 print(l)
3 print(type(l))
```

```
[]
<class 'list'>
```

2. Creating a list using list() function

You can create a list by using list() function. Generally, we use this function for creating lists from other data types like set, tuple etc…

> Syntax :
> `list_name = list(elements)`

PS. We already discussed earlier in Data Type Chapter Lesson 4 - Data Type Conversion.

Ex 9.2 (Creating a list using list() function)

```
l = list()
print(l)
print(type(l))
```

```
[]
<class 'list'>
```

💡 *Suppose you want to create a list of multiples of 3 between 5 and 40. How could you do it? Let us see..*

3. Creating list using list() & range() functions

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

> Syntax :
> `list_name = list(range(start, stop, step))`

- **Start** - Optional. An integer no. specifying at which position to start. Default is 0.
- **Stop** - Required. An integer no. specifying at which position to stop (not included).
- **Step** - Optional. An integer no. specifying the incrementation. Default is 1.

Ex 9.3 (Creating list using list() & range() functions)

```
M3 = list(range(6, 40, 3))

print(M3)
print(type(M3))
```

```
[6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39]
<class 'list'>
```

💡 *You want to go super market and buy, what will you do first. You will think what you want to buy then make a list of items.*

Let us consider you want to buy the following items (bag, pen, cap and another pen).

Ex 9.4 (Creating a list with elements)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print("my list:",my_list)
```

```
my list: ['bag', 'pen', 'cap', 'pen']
```

You can observe from the output that the order is preserved and also duplicate items are allowed.

PS. List can also contain mixed data type (strings, integers, boolean, None) elements.

While you list down items, you index goods from 0 to last item no. based on their priority (urgency).

💡 *What if you want to look for item written at no. 3?*

Lesson Two : Accessing a List

How to access a List?

There are three ways in which you can access the elements in the list. They are:

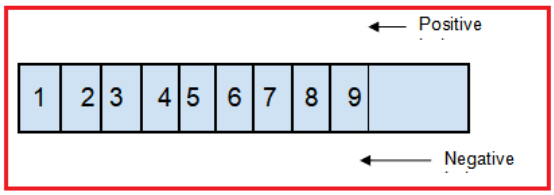
- By using indexing
- By using slicing operator
- By using loops (*PS. we will discuss in later Control Flow Chapter*)

1. Accessing a List Using Indexing

Indexing is the way of pointing to elements of a list.

> Syntax :
> `list_name[index_number]`

Accessing the elements by index means, accessing by their position numbers in the list. Index starts from 0 onwards. List supports both positive and negative indexes. Positive index represents from left to right direction and negative index represents from right to left direction



Ex 9.5 (List Indexing)

```
1 my_list = ['bag', 'pen', 'cap', 'pen']
2
3 print("my list:",my_list)
4 print("item at 3:",my_list[3])
5 print("item at -3:",my_list[-3])
6 print(type(my_list))
7 print(type(my_list[3]))
```

```
my list: ['bag', 'pen', 'cap', 'pen']
item at 3: pen
item at -3: pen
<class 'list'>
<class 'str'>
```

PS. For sake of simplicity, from now onward i means index number.

Note: If we are trying to access beyond the range of list index, then we will get IndexError.

Ex 9.6 (List index out of range error)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print(my_list[10])
```

```
---------------------------------------------------------
--------------------
IndexError                                Traceback
(most recent call last)
Cell In[6], line 2
      1 ml = ['bag', 'pen', 'cap', 'pen']
----> 2 print(ml[10])

IndexError: list index out of range
```

`EXERCISE 9.1`

`Create a list of your email, name, gender and age, then print your name?`

```
# write your answer code here

```

💡 *What if you want to look for goods listed from number 1 to 3 or listed on even numbers?*

2. Accessing List Using Slicing Operator

Slicing is the way of extracting a sublist (selected elements of a list) from the main list.

> Syntax :
> `list_name[start : stop : step]`

- **start** represents the index from where we slice should start, default value is 0.
- **stop** represents the end index where slice should end.
    - Max value allowed is last index + 1
    - If no value is provided, then it takes the last index of the list as default value.
- **step** represents the increment in the index position of the two consecutive elements to be sliced.
- The result of the slicing operation on the list is also a list i.e. it returns a list.

Ex 9.7 (List Slicing)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print("items from 1 to 3:",my_list[1:4])
print("odd index items:",my_list[1::2])
```

```
items from 1 to 3: ['pen', 'cap', 'pen']
odd index items: ['pen', 'pen']
```

EXERCISE 9.2

Create a list of your email, name, gender and age, then print your name and age?

```
# write your answer code here

```

💡 *What if you want to replace cap with hat?*

How to modify specific element of a list?

We can modify (reassign) specific element of a list.

> Syntax :
> `list_name[i] = new element`

Ex 9.8 (List Mutability)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print("Old list :",my_list)
my_list[2]= 'hat'
print("Modified list",my_list)
```

```
Old list : ['bag', 'pen', 'cap', 'pen']
Modified list ['bag', 'pen', 'hat', 'pen']
```

💡 *What if you want to know index of cap and pen?*

3. index() method

The index() method returns the position at the first occurrence of the specified item (value).

> Syntax :
> `list_name.index(element)`

Ex 9.9 (index() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print("cap index",my_list.index('cap'))
print("pen index",my_list.index('pen'))
```

```
cap index 2
pen index 1
```

4. enumerate() method

The enumerate() method returns pair of value and index position of all items in a list.

> Syntax :
> `list(enumerate(list_name)`

Ex 9.10 (List Mutability)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print(list(enumerate(my_list)))
```

```
[(0, 'bag'), (1, 'pen'), (2, 'cap'), (3, 'pen')]
```

PS. In python, a method and a function are one and the same. There is not much difference between them. It will be clearly explained later in OOPS chapter. For now just remember function and method are the same.

`EXERCISE 9.3`

`Create a list of your email, name, gender and age, then update your email and print index of all elements in the list?`

```
# write your answer code here

```

💡 *Ohhh... you forgot to write "oil" on your list, don't worry you can add to the list. How? Let us see..*

Lesson Three : Insertion of List Elements

1. append() method

Using this method we can add elements to the list. The elements will be added at the end of the list.

> Syntax :
> `list_name.append(new_element)`

Ex 9.11 (append() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print("Old list",my_list)
my_list.append("oil")
print("New list",my_list)
```

```
Old list ['bag', 'pen', 'cap', 'pen']
New list ['bag', 'pen', 'cap', 'pen', 'oil']
```

💡 *Here is the problem, "oil" is listed at end of the list, but you are starving now and you want to cook food as soon as you reach to your home. So "oil" should be listed first. How? Let us see...*

2. insert() method

The insert() method add the elements to the list at the specified index (position). It takes two arguments as input: one is the index and the other is the element.

> Syntax :
> `list_name.insert(i, new_element)`

Ex 9.12 (insert() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print("Old list",my_list)
ml.insert(0, "oil")
print("New list",my_list)
```

```
Old list ['bag', 'pen', 'cap', 'pen']
New list ['oil', 'bag', 'pen', 'cap', 'pen']
```

Difference between append() and insert()

Both the methods are used to add elements to the list. But append() will add the elements to the last, whereas insert() will add the elements at the specified index mentioned.

While you are using insert() method,

- If the specified index is greater than max index, then element will be inserted at last position.
- If the specified index is smaller than min index, then element will be inserted at first position.

`EXERCISE 9.4`

`Create a list of your email, name, gender and age, then append your password and insert your username to the list at index 1?`

```
# write your answer code here

```

💡 *Here is another issue, your father also want to buy goods but he is too busy so he asks you to buy him items. Therefore, you have to merge your list with his list? How? Let us see…*

3. extend() method

Using this method the items in one list can be added to the other.

> Syntax :
> `new_list.extend(old_list)`

Ex 9.13 (extend() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']
dad_list = ['tie', 'comb']

print('my list:', my_list)
print('my dad list:', dad_list)
my_list.extend(dad_list)
print('my list and my dad list:')
print(my_list)
```

```
my list: ['bag', 'pen', 'cap', 'pen']
my dad list: ['tie', 'comb']
my list and my dad list:
['bag', 'pen', 'cap', 'pen', 'tie', 'comb']
```

💡 *Suppose your mother also order you to pick her goods from store then you want to merge all lists: my_list, mom_list and dad_list, Let us see…*

Ex 9.14 (extend() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']
mom_list = ['oil', 'dye']
dad_list = ['tie', 'comb']

print('my list:', my_list)
print('my mom list:', mom_list)
print('my dad list:', dad_list)
my_list.extend(mom_list)
my_list.extend(dad_list)
print('my, mom and my dad list:')
print(my_list)
```

```
my list: ['bag', 'pen', 'cap', 'pen']
my mom list: ['oil', 'dye']
my dad list: ['tie', 'comb']
my, mom and my dad list:
['bag', 'pen', 'cap', 'pen', 'oil', 'dye', 'tie', 'comb']
```

💡 *But, Why you extend the list twice, Since Number of lines in a program is important for code efficiency / execution time /, Why don't we complete it within a single line? Let us see…*

4. Concatenation operator (+)

"+" operator concatenates two list objects to join them and returns a single list.

PS. For this, both the operands should be list type, else we will get TypeError.

> Syntax :
> `list_3 = list_1 + list_2`

Ex 9.15 (Concatenation (+) operator)

```
my_list = ['bag', 'pen', 'cap', 'pen']
mom_list = ['oil', 'dye']
dad_list = ['tie', 'comb']

print('my list:', my_list)
print('my mom list:', mom_list)
print('my dad list:', dad_list)
my_list = my_list + mom_list + dad_list
print('my, mom and my dad list:')
print(my_list)
```

```
my list: ['bag', 'pen', 'cap', 'pen']
my mom list: ['oil', 'dye']
my dad list: ['tie', 'comb']
my, mom and my dad list:
['bag', 'pen', 'cap', 'pen', 'oil', 'dye', 'tie', 'comb']
```

Ex 9.16 (Concatenation (+) operator TypeError)

```
a = [1, 2, 3]
b = 'pen'

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

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
Cell In[17], line 3
      1 a = [1, 2, 3]
      2 b = 'pen'
----> 3 c = a + b
      4 print(c)

TypeError: can only concatenate list (not "str") to
```

EXERCISE 9.5

Create a list of your email, name, gender and age, then create another list which contains your birth date, month, year and extend/add it to former list?

```
# write your answer code here

```

💡 *Hell no… on your way your sister called you then told you that he found your pen so you want to remove one pen from your list. How? Let us see…*

Lesson Four : Deletion of a List

There are many ways of deleting & clearing elements from a list. Let's try to understand each of them.

1. remove() method

This method used to remove specific item from a list.

> Syntax :
> `list_name.remove(element)`

Ex 9.17 (remove() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Old list:', my_list)
my_list.remove('pen')
print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: ['bag', 'cap', 'pen']
```

- If the item exists multiple times, then only the first occurrence will be removed.
- If the specified item not present in list, then we will get ValueError.
- Before removing elements it's a good approach to check if the element exists or not to avoid errors.

Ex 9.18 (remove() method ValueError)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Old list:', my_list)
my_list.remove('hat')
print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
---------------------------------------------------------
--------------------
ValueError                                Traceback
(most recent call last)
Cell In[19], line 3
      1 ml = ['bag', 'pen', 'cap', 'pen']
      2 print('Old list:', ml)
----> 3 ml.remove('hat')
      4 print('New list:', ml)

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

💡 *What if you want to delete the last indexed pen?*

2. pop() method

This method takes an index of removable element.

- If no index is given, then the last item is removed and returned by default.
- If the provided index is not in the range, then IndexError is thrown.

> Syntax :
>
> `list_name.pop(i)`

Ex 9.19 (pop() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Old list:', my_list)
my_list.pop(3)
print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: ['bag', 'pen', 'cap']
```

💡 *While you are purchasing in supermarket you are running out of money so what will you do? You should remove item based on its priority so last item should be removed. How? Let us see…*

Ex 9.20 (pop() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Old list:', my_list)
my_list.pop()
print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: ['bag', 'pen', 'cap']
```

Ex 9.21 (pop() method IndexError)

```
my_list = ['bag', 'pen', 'cap', 'pen']

my_list.pop(10)
```

```
---------------------------------------------------------
--------------------
IndexError                                Traceback
(most recent call last)
<ipython-input-2-858b5e8b67d2> in <cell line: 2>()
      1 my_list = ['bag', 'pen', 'cap', 'pen']
----> 2 my_list.pop(10)

IndexError: pop index out of range
```

Difference between remove() and pop()

remove()	pop()
To remove based on the element.	To remove based on index
It doesn't return any value	It returns the popped out element.
If the item is not present, the we get ValueError	If the index is not in the range, then we get IndexError

💡 *What if you want to delete selected items?*

3. del statement

The del statement can be used to delete an element at a given index. Also, it can be used to remove slices from a list.

> Syntax :
> `del list_name[start : stop : step]`

Ex 9.22 (del statement / sliced)

```
1 my_list = ['bag', 'pen', 'cap', 'pen']
2
3 print('Old list:', my_list)
4 del my_list[1:3]
5 print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: ['bag', 'pen']
```

Ex 9.23 (del statement / sliced)

```
1 my_list = ['bag', 'pen', 'cap', 'pen']
2
3 print('Old list:', my_list)
4 del my_list[1:2:]
5 print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: ['bag', 'cap', 'pen']
```

💡 *What if you want to delete all items?*

Ex 9.24 (del statement / all)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Old list:', my_list)
del my_list[:]
print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: []
```

4. clear() method

clear() method removes all the elements from a list.

> Syntax :
> `list_name.clear()`

Ex 9.25 (clear() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Old list:', my_list)
my_list.clear()
print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: []
```

5. *= method

This method removes all the elements by selecting all elements of a list then truncate them from the list.

> Syntax :
> `list_name *= 0`

Ex 9.26 (*= method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Old list:', my_list)
my_list *= 0
print('New list:', my_list)
```

```
Old list: ['bag', 'pen', 'cap', 'pen']
New list: []
```

EXERCISE 9.6

Create a list of your email, name, gender and age, then if the email is not gmail delete it, if the gender is male delete all elements from the list?

```
# write your answer code here

```

💡 *What if your list contains large number of items. It is difficult to find and manage your items. So you need to sort them based on alphabetical order (A to Z). How? Let us see…*

Lesson Five : Arranging a List

1. sort() method

In a list by default insertion order is preserved. If we want to sort the elements of the list according to default natural sorting order then we should go for sort() method.

- For numbers the default natural sorting order is ascending order.
- For strings the default natural sorting order is alphabetical order.
- To use sort() method, list should contain only homogeneous elements, otherwise we will get TypeError.

> Syntax :
> `list_name.sort()`

Ex 9.27 (sort() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Unsorted:', my_list)
my_list.sort()
print('Sorted:', my_list)
```

```
Unsorted: ['bag', 'pen', 'cap', 'pen']
Sorted: ['bag', 'cap', 'pen', 'pen']
```

Ex 9.28 (sort() method TypeError)

```
n = [4, 5, 1, 'bag', 2, 'pen']

n.sort()
print(n)
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
<ipython-input-9-dffd5406ba98> in <cell line: 2>()
      1 n = [4, 5, 1, 'bag', 2, 'pen']
----> 2 n.sort()
      3 print(n)

TypeError: '<' not supported between instances of 'str'
```

💡 *What if you want to reverse order of items?*

2. reverse() method

This method reverses the order of list elements.

> Syntax :
> `list_name.reverse()`

Ex 9.29 (reverse() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']

print('Unsorted list:', my_list)
my_list.reverse()
print('Reverse sort list:', my_list)
```

```
Unsorted list: ['bag', 'pen', 'cap', 'pen']
Reverse sort list: ['pen', 'cap', 'pen', 'bag']
```

`EXERCISE 9.7`

`Create a list of name of grade 10 students then sort them from A to Z and Z to A?`

```
# write your answer code here

```

💡 *Earlier we use extend() function to merge two lists? But Now we want to categorize elements of a list. Here elements of a list are treated as a list themselves.*

💡 *What if you want to group items based on their type (Electronics, Stationary, Food)?*

3. Nested Lists

A list within another list is called a nested list. It is possible to take a list as an element in another list.

> Syntax :
> `list_name=[[elements..], [..]..]`

Ex 9.30 (Nested Lists)

```
my_list = [['oat', 'egg'], ['bag', 'pen']]

print('my list:', my_list)
print('Food list:', my_list[0])
print('Stationary list:', my_list[1])
print('1st Food item:',my_list[0][0])
```

```
my list: [['oat', 'egg'], ['bag', 'pen']]
Food list: ['oat', 'egg']
Stationary list: ['bag', 'pen']
1st Food item: oat
```

EXERCISE 9.8

Create a list of name of grade 10 students then classify them as Section 10A and Section 10B students?

```
# write your answer code here

```

💡 *Suppose, you have twin and your twin also want to buy exactly the same items that you want to buy, but in different supermarket so how would you copy your item list to her?*

Lesson Six : List Aliasing and List Cloning

What is List Aliasing?

The process of giving a new name to an existing one is called Aliasing.

- The new name is called an alias name. Both names will refer to the same memory location.
- If we do any modifications on the data, then it will be updated at the memory location.
- Since both the actual and alias name refer to the same location, any modifications done on an existing object will be reflected in the new one also.

> Syntax :
> `list_1 = list_2`

PS. list_1 is new list (alias name) and list_2 is original list (actual name).

Ex 9.31 (List Aliasing)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = my_list

print('my list:', my_list)
print('twin list:', twin_list)
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: ['bag', 'pen', 'cap', 'pen']
```

💡 *What if your twin want to buy book instead of bag because she already have one? so what will you do?*

Ex 9.32 (List Aliasing)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = my_list

twin_list[0] = 'book'
print('my list:', my_list)
print('twin list:', twin_list)
print('Memory Address of my list',id(my_list))
print('Memory Address of twin list',id(twin_list))
```

```
my list: ['book', 'pen', 'cap', 'pen']
twin list: ['book', 'pen', 'cap', 'pen']
Memory Address of my list 134125662939520
Memory Address of twin list 134125662939520
```

💡 *Here is the problem. your twin want to buy book but you want to buy a bag. Solve this? Let's see…*

PS. In List Aliasing, memory address of my_list and twin_list was the same. What if we have different memory address for each list?

What is List Cloning?

The process of creating duplicate independent objects is called cloning. We can implement it

1. by using extend() method
2. by using concatenation (+)
3. by using slice operator
4. by using list() method
5. by using copy() method
6. by using method of Shallow copy
7. by using method of Deep Copy
8. by using append() method
9. by using List Comprehension

These processes create a duplicate of the existing one at a different memory location. so, both objects will be independent, and applying any modifications on one will not impact the other.

💡 *Earlier, we use extend() method to merge two lists, Now we can use to copy a list. How? Let us see..*

- First, we create an empty list
- Then, we merge the empty list to original list

1. List Cloning using extend() method

The lists can be copied into a new list by using the extend() function. This appends each element of the original list to the end of the new list.

> Syntax :
> `list_1 = [] list_1.extend(list_2)`

Ex 9.33 (Cloning using extend() method)

```
1 my_list = ['bag', 'pen', 'cap', 'pen']
2 twin_list = []
3
4 twin_list.extend(my_list)
5 twin_list[0] = 'book'
6 print('my list:', my_list)
7 print('twin list:', twin_list)
8 print('Memory Address of my list:', id(my_list))
9 print('Memory Address of twin list:', id(twin_list))
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: ['book', 'pen', 'cap', 'pen']
Memory Address of my list: 134125664932544
Memory Address of twin list: 134125662967232
```

2. List Cloning using concatenation (+)

Concatenation (+) operator appends each element of the List to the end of the new list.

> Syntax :
> `list_1 = [] list_1 += list_2`

Ex 9.34 (Cloning using extend() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = []

twin_list += my_list
twin_list[0] = 'book'

print('my list:', my_list)
print('twin list:', twin_list)
print('Memory Address of my list:', id(my_list))
print('Memory Address of twin list:', id(twin_list))
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: ['book', 'pen', 'cap', 'pen']
Memory Address of my list: 134124835512128
Memory Address of twin list: 134124833699008
```

💡 *What if you want to copy specific elements of a list? Suppose your twin want to buy only book and cap. you can create another new twin list, but why don't we clone your list since it contains all items (except "book") that your twin want to buy? How could you do it? Let us see…*

💡 *Earlier, we use slice (:) operator to slice then access elements of a list, Now we can use to copy a list. How? Let us see…*

- First, we slice all or specific elements of original list
- Then, we create a new list using sliced elements

3. List Cloning using slice operator

This method is considered when we want to modify a list and also keep a copy of the original. In this we make a copy of the list itself, along with the reference.

> Syntax :
> `list_1 = list_2[start:stop:step]`

Ex 9.35 (Cloning using slicing operator)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = my_list[0:4:2]

twin_list[0] = 'book'
print('my list:', my_list)
print('twin list:', twin_list)
print('Memory Address of my list:', id(my_list))
print('Memory Address of twin list:', id(twin_list))
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: ['book', 'cap']
Memory Address of my list: 134125664854016
Memory Address of twin list: 134125664902080
```

💡 *Earlier, we use list() method to create a list, Now we use to copy a list. How? Let us see…*

- First, we try to access all elements of original list
- Then, we create a new list using list() function including accessed elements

4. List Cloning using list() method

This method of cloning uses a built-in function list().

> Syntax :
> `list_1 = list(list_2)`

Ex 9.36 (Cloning using list() method)

```
1 my_list = ['bag', 'pen', 'cap', 'pen']
2 twin_list = list(my_list)
3
4 twin_list[0] = 'book'
5 print('my list:', my_list)
6 print('twin list:', twin_list)
7 print('Memory Address of my list:', id(my_list))
8 print('Memory Address of twin list:', id(twin_list))
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: ['book', 'pen', 'cap', 'pen']
Memory Address of my list: 134125661681152
Memory Address of twin list: 134124835507520
```

5. List Cloning using copy() method

The inbuilt method copy is used to copy all the elements from one list to another.

> Syntax :
> `list_1 = list_2.copy()`

Ex 9.37 (Cloning by using copy() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = my_list.copy()

twin_list[0] = 'book'
print('my list:', my_list)
print('twin list:', twin_list)
print('Memory Address of my list:', id(my_list))
print('Memory Address of twin list:', id(twin_list))
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: ['book', 'pen', 'cap', 'pen']
Memory Address of my list: 134124833691456
Memory Address of twin list: 134125656410368
```

6. List Cloning ~ method of Shallow Copy

A shallow copy creates a new List which stores the reference of the original list elements.

A **shallow copy** means constructing a new collection list and then populating it with references to the elements of list found in the original.

> Syntax :
> `import copy`
> `list_1 = copy.copy(list_2)`

Ex 9.38 (Cloning using method of Shallow Copy)

```
import copy
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = copy.copy(my_list)

twin_list[0] = 'book'
print('my list:', my_list)
print('twin list:', twin_list)
print('Memory Address of my list:', id(my_list))
print('Memory Address of twin list:', id(twin_list))
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: ['book', 'pen', 'cap', 'pen']
MA of my list: 2656157389888
MA of twin list: 2656157396992
```

`EXERCISE 9.9`

`Create a list of name of grade 10 students then add extra student ‘John’ then copy the list?`

```
# write your answer code here

```

PS. Shallow copy doesn't create a copy of nested elements of list, instead it just copies the reference of nested list. This means, a copy process does not recurse or create copies of nested elements of a list.

💡 *Here is the problem, even though we use the above six cloning methods (including method of Shallow Copy) to copy nested list, we couldn't modify an element independently. Let us check it…*

Ex 9.39 (Nested List Cloning using Shallow Copy)

```
import copy
my_list = [['oat', 'egg'],['bag', 'pen']]
twin_list = copy.copy(my_list)

twin_list[1][0] = 'book'
print('my list:', my_list)
print('twin list:', twin_list)
print('Memory Address of my list:', id(my_list))
print('Memory Address of twin list:', id(twin_list))
```

```
my list: [['oat', 'egg'], ['book', 'pen']]
twin list: [['oat', 'egg'], ['book', 'pen']]
MA of my list: 2656157285760
MA of twin list: 2656157308736
```

💡 *Therefore, we need to use method of deep copy for cloning nested list.*

7. List Cloning using method of Deep Copy

Deep copy creates a new list and recursively adds the copies of nested elements present in the list.

> Syntax :
> `import copy`
> `list_1 = copy.deepcopy(list_2)`

Ex 9.40 (Cloning by using copy() method)

```
import copy
my_list = [['oat','egg'],['bag','pen']]
twin_list = copy.deepcopy(my_list)

twin_list[1][0] = 'book'
print('my list:', my_list)
print('twin list:', twin_list)
print('Memory Address of my list:', id(my_list))
print('Memory Address of twin list:', id(twin_list))
```

```
my list: [['oat', 'egg'], ['bag', 'pen']]
twin list: [['oat', 'egg'], ['book', 'pen']]
Memory Address of my list: 134125663881920
Memory Address of twin list: 134125664520384
```

8. List Cloning using List Comprehension

The method of list comprehension can be used to copy all the elements individually from one list to another.

PS. We will see later in Control Flow Chapter.

9. List Cloning using append() method

This can be used for appending and adding elements to list or copying them to a new list. It is used to add elements to the last position of list.

PS. We will see later in Control Flow Chapter.

`EXERCISE 9.11`

`Create a nested list of name of grade Section 10A and Section 10B students then add extra student ‘John’ then copy the list?`

```
# write your answer code here

```

Lesson Seven : Mathematical Operations on List (using List Methods)

💡 *What if you want to know total number of items listed to buy?*

1. len() method

len() function returns the number (length) of elements in a list.

> Syntax :
> `len(list_name)`

Ex 9.41 (len() function)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = [['oat', 'egg'], ['bag', 'pen']]

print("my list length:",len(my_list))
print("twin list length:",len(twin_list))
```

```
my list length: 4
twin list length: 2
```

💡 *What if you want to know how many pens you listed to buy?*

2. count() method

This method returns the number of occurrences of specific item in the list.

> Syntax :
> `list_name.count(element)`

Ex 9.42 (count() method)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = [['oat', 'egg'], ['bag', 'pen']]

print("Pen occurs:",my_list.count('pen'))
print("Pen occurs:",twin_list.count('pen'))
```

```
Pen occurs: 2
Pen occurs: 0
```

3. max()

max() function returns the largest element of a list.

> Syntax :
> `max(list_name)`

Ex 9.43 (max() method)

```
n = [7, 6, 9, 2, 3]
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = [['oat', 'egg'], ['bag', 'pen']]

print("max of numbers list:",max(n))
print("max of my list:",max(my_list))
print("max of twin list:",max(twin_list))
```

```
max of numbers list: 9
max of my list: pen
max of twin list: ['oat', 'egg']
```

4. min()

min() function returns the smallest element of a list.

> Syntax :
> `min(list_name)`

Ex 9.44 (min() method)

```
n = [7, 6, 9, 2, 3]
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = [['oat', 'egg'], ['bag', 'pen']]

print("min of numbers list:",max(n))
print("min of my list:",max(my_list))
print("min of twin list:",max(twin_list))
```

```
min of numbers list: 9
min of my list: pen
min of twin list: ['oat', 'egg']
```

5. random.choice() method

Sometimes, while working with dictionaries, we can have a situation in which we need to find a random pair from dictionary. This type of problem can come in games such as lotteries etc. Let’s discuss certain ways in which this task can be performed.

> Syntax:
> `import random`
> `random.choice(list_name)`

Ex 9.45 (random(choice()) method)

```
import random
players = {'Messi', 'Ronaldo', 'Salah'}

print("The Winner is " ,end="")
print(random.choice(list(players)))
```

```
The Winner is Salah
```

EXERCISE 9.12

Create a list in range 1 to 10 first the computer picks one number and you guess that number then if your guess is correct, you win else you lose?

```
# write your answer code here

```

6. sum()

sum() function adds only integer elements of a list and returns the sum.

> Syntax :
> `sum(list_name)`

Ex 9.46 (sum() method)

```
n = [7, 6, 9, 2, 3]

print("Sum of numbers list:",sum(n))
```

```
Sum of numbers list: 27
```

PS. max(), min() and sum() functions only works for homogenous data type elements, else your code will return an error.

EXERCISE 9.13

Create a list of random five numbers then check whether there is duplicated number or not if it is there delete it, if sum of all numbers is even print maximum of the list else print minimum of the list.

```
# write your answer code here

```

Lesson Eight : Mathematical Operations on on List (using List Operators)

💡 *What if you want to duplicate all items of a list in the list itself?*

1. Multiplication operator (*)

The " * " operator works to repeat elements in the list by the said number of times. For this one operand should be list and the other operand should be integer, else we get TypeError.

> Syntax :
> `no. of repeat * list_name`

Ex 9.47 (Multiplication (*) operator)

```
mom_list = ['oil', 'dye']

print('Old list:', mom_list)
print('New list:', (2*mom_list))
```

```
Old list: ['oil', 'dye']
New list: ['oil', 'dye', 'oil', 'dye']
```

💡 *In Operators Chapter we compared variables of different data types. Now we are going to compare two Lists? How? Let us see…*

2. Relational Operators

We can use relational operators (<, <=,>,>=) on the List elements for comparing two lists.

- Initially the first two items are compared, and if they are not the same then the corresponding result is returned.
- If they are equal, the next two items are compared, and so on

Ex 9.48 (Comparison Operators)

```
print([1, 2, 3] < [2, 2, 3])
print([1, 2, 3] <= [1, 2, 3])
print([1, 2, 3, 4] > [1, 6, 3])
print([1, 2, 3, 4] >= [1, 6, 3])
print([1, 2, 1] == [4])
```

```
True
True
False
False
False
```

While comparing lists which are loaded with strings the following things are considered for the comparison.

- The number of elements
- The order of elements
- The content of elements (case sensitive)

Ex 9.49 (Comparison of lists)

```
x = ["abc", "def", "ghi"]
y = ["abc", "def", "ghi"]
z = ["ABC", "DEF", "GHI"]
a = ["abc", "def", "ghi", "jkl"]
print(x < z)
print(a < x)
print(x == y)
print(x == z)
print(x == a)
```

```
False
False
True
False
False
```

💡 *What if you want to check whether "book" is in your list or not?*

💡 *What if you want to check whether "oat" is in your twin nested list or not?*

3. Membership Operators

We can check if the element is a member of a list or not by using membership operators. They are:

- **in operator** : If the element is a member of list, then the in operator returns True otherwise False.
- **not in operator** : If the element is not in the list, then not in operator returns True otherwise False.

Ex 9.50 (Membership operators)

```
my_list = ['bag', 'pen', 'cap', 'pen']
twin_list = [['oat', 'egg'], ['bag', 'pen']]

print('my list:', my_list)
print('twin list:', twin_list)
print('bag' in my_list)
print('Bag' not in my_list)
print('book' in my_list)
print('oat' in twin_list)
```

```
my list: ['bag', 'pen', 'cap', 'pen']
twin list: [['oat', 'egg'], ['bag', 'pen']]
True
True
False
False
```

EXERCISE 9.14

Create a list of random three numbers 1 to 10, then if 2 is in the list duplicate the list else delete the list?

```
# write your answer code here

```

Lesson Nine : Introduction to Tuple

What are Tuples?

A tuple refers to a collection of objects, which are ordered and can't be changed. These objects can be strings, lists, tuples, integers etc.

Characteristics and Features of Tuples

- **Insertion order is preserved:** The order in which elements are added to the tuple is the order in which the output will be displayed.
- **Immutable**: Once if we create a tuple object, then we cannot modify the content of the tuple object. This is the only feature which differentiates tuples from the lists.
- Tuple can **store duplicates**.
- Tuple can store the same type of objects and different types of objects as well.

When should we go for tuple data structure?

If we are going to define a data which is not going to change over the period, then we should go for tuple data structure. For example, week days, calendar months, years etc.

How to create a tuple?

Different Ways to Create Tuples:

- Empty Tuple
- Single value tuple
- Tuple with group of values
- By using tuple() function

1. Empty Tuple

We can create an empty tuple by using empty parenthesis.

> Syntax :
> `tuple_name = ()`

Ex 9.51 (Empty Tuple)

```
1 emp = ()
2
3 print(emp)
4 print(type(emp))
```

```
()
<class 'tuple'>
```

2. Single Value Tuple

If a tuple has only one object, then that object should end with a comma separator otherwise python, internally, will not consider it as a tuple.

Ex 9.52 (Single Value Tuple)

```
1 name = ("John")
2 a = (11)
3
4 print(name)
5 print(a)
6 print(type(name))
7 print(type(a))
```

```
John
11
<class 'str'>
<class 'int'>
```

We can create a single value tuple in two ways:

- Using parenthesis – Comma is mandatory
- Without using parenthesis – Comma is mandatory

> Syntax :
> `tuple_name = (element,)`

Ex 9.53 (Single Value Tuple)

```
1 name = ("John",)
2 a = (11,)
3
4 print(name)
5 print(a)
6 print(type(name))
7 print(type(a))
```

```
('John',)
(11,)
<class 'tuple'>
<class 'tuple'>
```

3. Tuple with group of values

Tuple can contain group of objects; those objects can be of the same type or different types. While creating tuple with group of values parentheses are optional.

> Syntax :
> `tuple_name = (elements)`

Ex 9.54 (Tuple with group of values)

```
1 weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
2
3 print("Weekdays :",weekdays)
4 print(type(weekdays))
```

```
Weekdays : ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
<class 'tuple'>
```

PS. parenthesis are optional, when creating tuples.

Ex 9.55 (Parenthesis is optional for tuple)

```
weekdays = "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"

print("Weekdays :",weekdays)
print(type(weekdays))
```

```
Weekdays : ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
<class 'tuple'>
```

4. By using tuple() function

We can create a tuple by using tuple() function.

> Syntax :
> `tuple_name = tuple(list_name)`

Ex 9.56 (Using Tuple Function)

```
my_list = [11, 22, 33]
my_tuple = tuple(my_list)

print(my_tuple)
```

```
(11, 22, 33)
```

> Syntax :
> `tuple_name = tuple(start, stop, step)`

Ex 9.57 (Using range() Function)

```
my_tuple = tuple(range(1, 10, 2))
print(my_tuple)
```

```
(1, 3, 5, 7, 9)
```

Tuple Immutability

Tuple having immutable nature. If we create a tuple then we cannot modify the elements of the existing tuple.

Ex 9.58 (Tuple Immutability)

```
my_tuple = (10, 20, 30, 40)

my_tuple[1] = 70
```

```
------------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
<ipython-input-38-11caf04c28a2> in <cell line: 2>()
      1 my_tuple = (10, 20, 30, 40)
----> 2 my_tuple[1] = 70

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

sorted() function in Tuple

This function can sort the elements which is default natural sorting order. This returns a sorted list of elements.

> Syntax :
>
> `new_tuple = sorted(tuple_name)`

Ex 9.59 (sorted() method in Tuples)

```
1 my_tuple = (40, 10, 30, 20)
2 sorted_tuple = sorted(my_tuple)
3 print(my_tuple)
4 print(sorted_tuple)
```

```
(40, 10, 30, 20)
[10, 20, 30, 40]
```

We can sort according to reverse of default natural sorting order as below

> Syntax :
>
> `new_tuple = sorted(tuple_name, reverse=True)`

Ex 9.60 (sorted() method in Tuples)

```
1 my_tuple = (40, 10, 30, 20)
2 sorted_tuple = sorted(my_tuple, reverse = True)
3 print(sorted_tuple)
```

```
[40, 30, 20, 10]
```

Tuple Packing

We can create a tuple by packing a group of variables.

> Syntax :
>
> `tuple_name = elements`

Ex 9.61 (Tuple packing)

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

```
(10, 20, 30, 40)
```

Tuple Unpacking

Tuple unpacking is the reverse process of tuple packing, we can unpack a tuple and assign its values to different variables.

> Syntax :
> `elements = tuple_name`

Ex 9.62 Tuple unpacking

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

```
a = 10  b = 20  c = 30  d = 40
```

Note: At the time of tuple unpacking the number of variables and number of Values should be the same, otherwise you will get ValueError.

Difference between List and Tuple

List and Tuple are exactly the same except for a small difference: List objects are mutable Whereas Tuple objects are immutable.

List	Tuple
List is a Group of Comma separated Values within Square Brackets and Square Brackets are mandatory. **Example**: i = [10, 20, 30,40]	Tuple is a Group of Comma separated Values within Parentheses and Parenthesis are optional. **Example**: t = (10, 20, 30,40) **Example**: t = 10, 20, 30,40
List Objects are Mutable i.e.once we create a list object we can perform any changes in that.	Tuple Objects are Immutable i.e. once we create a tuple object we cannot change its content. If we try to do any changes then we get errors.
If the Content is not fixed and keeps on changing, then we should go for list data type.	If the content is fixed and never changes then we should go for Tuples.
List Objects cannot be used as keys for dictionaries because keys should be Hash table and Immutable.	Tuple Objects can be used as keys for dictionaries because keys should be Hash - able and Immutable

EXERCISE 9.14

Create a tuple of random three numbers 1 to 10, then if 2 is in the tuple sort the tuple else reverse sort then unpack the tuple?

```
1 # write your answer code here
2
```

Key Takeaways

- *Data Structure is a particular way of organizing data in a computer.*
- *A list is a data structure which can store a group of elements or objects in a single variable.*
- *You can access elements of a list using indexing, slicing, or loops.*
- *You can insert elements in a list using* `append(), insert(), extend()` *methods.*
- *You can delete elements of list using* `remove(), pop(), clear(), *=` *methods and* `del` *statement.*
- *You can arrange elements of a list using* `sort()` *and* `reverse()` *methods.*
- *A list within another list is called a nested list.*
- *The process of giving a new name to an existing one is called Aliasing.*
- *The process of creating duplicate independent objects is called cloning.*
- *You can use operators (*, membership and relational operators) and methods (* `len(), count(), max(), min(), random.choice(), sum()` *) on a list.*
- *Tuple is a collection of objects, which are ordered and can't be changed.*
- *List objects are mutable Whereas Tuple objects are immutable.*

// END OF CHAPTER NINE

Click here for Chapter Ten

CHAPTER TEN

Control Flow (Loops)

Objectives

- *What is a loop?*
- *What are the advantages of loops?*
- *What are the two types of loops in Python?*
- *What is an infinite loop?*
- *What is a nested loop?*
- *What are break, continue, and pass statements?*
- *What is the difference between pass and comment?*
- *What is list comprehension?*

Lesson One : Introduction to Loops

Introduction to Loops

Computers are great at repeating the same task over and over. They never get bored or make a mistake. You could ask a computer a thousand times in the first result would be just as accurate as the last, which isn't something we can say about us humans. Have you ever tried to do something at thousand times in a row, it be enough to drive you loopy.

The ability to accurately perform repetitive tasks and never get tired is why computers are so great for automation. It doesn't matter how complex the task is your computer will do it as many times as you tell it to.

Advantages of loops

- It provides code reusability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of sequence data types.

LOOPING (Iterations) Statement in Python

If we want to execute a group of statements in multiple times, then we should go for looping kind of execution.

💡 *What is loop?*

A loop is a construction that allows you to execute a block of code multiple times repeatedly until the program meets a certain given condition. It consists of two parts - a condition and a body.

Condition and body

► Loop condition

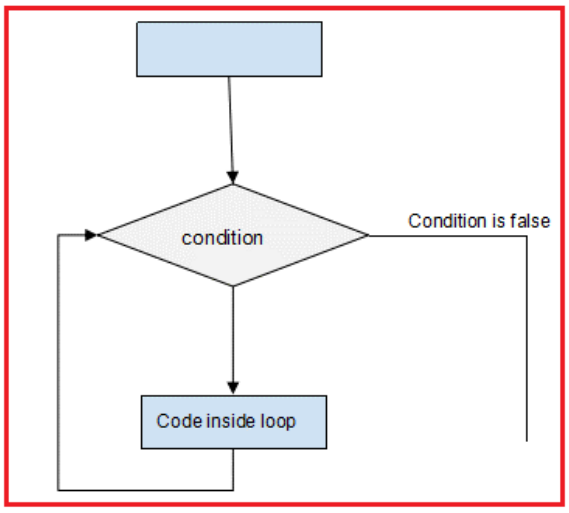
Every loop has a condition. This is the part that controls if the loop should continue or stop. All loops continue their iterations if the condition is true. The condition is any valid value or expression that could be evaluated to a value. A value of 0 or '\0' (null) is considered as "false" and everything else is "true"

► Body

The body is one or more valid statements. If it consists of more than one statement, the body must be enclosed in curly brackets { } .

💡 *How loops work?*

The loop starts by entering the condition. If it is "true" the body will be executed. Then the execution repeats the check of the condition and then the body. It looks like this:



begin -> condition(true) -> body -> condition(true) -> body ... condition(false) -> end.

There are two types of loops available in python. They are:

- while loop / waits till you fail /
- for loop / waits till you succeed /

while Loop

> Syntax for while loop :
> `Initialization`
> `while condition:`
> `    statements`
> `increment / decrement`

The while loop contains an expression/condition. As per the syntax colon (:) is mandatory otherwise it throws syntax error. Condition gives the result as bool type, either True or False. The loop keeps on executing the statements until the condition becomes False.

Parts of while loop

Initialization: This is the first part of the while loop. Before entering the condition section, some initialization are required.

Condition: Once the initializations done, then it will go for condition checking which is the heart of the while loop. The condition is checked and if it returns True, then execution enters the loop for executing the statements inside.

After executing the statements, the execution goes to the increment/decrement section to increment the iterator. Mostly, the condition will be based on this iterator value, which keeps on changing for each iteration. This completes the first iteration.

In the second iteration, if the condition is False, then the execution comes out of the loop else it will proceed as explained in the above point.

Increment/Decrement section: This is the section where the iterator is increased or decreased.

💡 *Suppose you have a folder that contains List of Songs: Michael Jackson Thriller Album, Then you want to play all music in the album. How? Let us see…*

PS. For sake of simplicity we consider that the folder contains only 4 songs from the album.

Ex 10.1 (while loop)

```
Music = 'Thriller'

i = 1
while i <= 3:
  print("Playing",Music)
  i+=1
```

```
Playing Thriller
Playing Thriller
Playing Thriller
```

Before starting the loop, we have made some assignments(i = 1). This is called the Initialization.

After initialization, we started the while loop with a condition i <= 3. This condition returns True until i is less than 3.

Inside the loop we are playing the music “Thriller”.

After the music played, we are incrementing the value of i using the operator i+=1. This is called the incrementation/decrementing.

For each iteration the value of i will increase and when the i value reaches 4, then the condition i<=3 returns False. At this iteration, the execution comes out of the loop without executing the statements inside. Hence the music is not played for 4th time.

EXERCISE 10.1

Print all multiples of 7 from 1 to 100 using while loop?

```
# write your answer code here

```

Ex 10.2 (while loop)

```
Album = ['WBSS', 'Thriller', 'Beat It', 'PYT']

i = 0
while i < len(Album):
  print("Playing",Album[i])
  i+=1
```

```
Playing WBSS
Playing Thriller
Playing Beat It
Playing PYT
```

EXERCISE 10.2

Print Fibonacci Sequence till 50?

A Fibonacci Sequence is a series of numbers where each number is the sum of the two numbers that precede it. 0, 1, 1, 2, 3, 5, 8, …

```
# write your answer code here

```

Lesson Two : for Loop

Basically, for loop is used to iterate elements one by one from sequences like string, list, tuple, etc. This loop can be easily understood when compared to while loop. While iterating elements from sequence we can perform operations on every element.

> Syntax of for loop :
> `for variable in sequence:`
> `    statements`

Only iterable data types are used in sequence, those are List, Tuple, Set, Dictionary and String.

💡 *Suppose you have a folder that contains List of Songs: Michael Jackson Thriller Album, Then you want to play all music in the album. How? Let us see…*

PS. For sake of simplicity we consider that the folder contains only 4 songs from the album.

Ex 10.3 (for loop)

```
Album = ['WBSS', 'Thriller', 'Beat It', 'PYT']

for i in Album:
  print("Playing",i)
```

```
Playing WBSS
Playing Thriller
Playing Beat It
Playing PYT
```

for Loop using range() Function

We can create a range of values by using range() function. The range data type represents a sequence of numbers. The range data type is immutable, means we cannot modify it once it is created.

> Syntax of for loop using range() function:
>
> ```
> for variable in sequence:
>     statements
> ```

EXERCISE 10.3

Find factorial (n!) of given number from user?

```
# write your answer code here

```

Lesson Three : Infinite and Nested Loop

💡 *Suppose you want to listen only one music throughout your life, How you achieve this? Let's see*

Infinite Loop

The loop which never ends, or the loop whose condition never gets False is called an infinite loop. This loop never exits. In order to come out of the loop we need to manually do it by command **`Ctrl + C`**.

> Syntax of infinite for loop :
>
> ```
> for variable in sequence:
>     statements
> ```

Ex 10.4 (Infinite for loop)

```
import itertools
Music = ['Thriller']

for i in itertools.count():
  print("Playing",Music)
```

```
Playing ['Thriller']
Playing ['Thriller']
. . .
```

EXERCISE 10.4

Print all perfect square numbers from 0 to 100?

```
# write your answer code here

```

> Syntax of infinite while loop :
>
> `while True:`
>
> `    statements`

Ex 10.5 (Infinite while loop)

```
Music = 'Thriller'

while True:
  print("Playing",Music)
```

```
Playing ['Thriller']
Playing ['Thriller']
Playing ['Thriller']
...
```

EXERCISE 10.5

Print all prime numbers from 0 to 100?

```
# write your answer code here

```

💡 *Suppose you want to play two albums (The weeknd and Michael Jackson) from music folder?*

Nested Loops

If a loop is written inside another loop then it is called a nested loop. Let's understand it with an example.

> Syntax of nested for loops :
>
> `for variable_1 in sequence:`
>
> `    statements /optional/`
>
> `    for variable_2 in sequence:`
>
> `        statements`

Ex 10.6 (Nested for Loop)

```
Music_Folder = [['D.D.', 'Outside'],
                ['WBBS', 'Thriller', 'Beat It']]

for i in Music_Folder:
  for j in i:
    print("Playing",j)
```

```
Playing D.D.
Playing Outside
Playing WBBS
Playing Thriller
Playing Beat It
```

> Syntax of nested while loops : `Initialization_1`
>
> ```
> while condition_1:
>     statements/optional/
>     initalization_2
>
>     while condition_2:
>         statements
>         Increment/Decrement_2
>
>     Increment/Decrement_1
> ```

Ex 10.7 (Nested while Loop)

```
1 Music_Folder = [['D.D.', 'Outside'],
2                 ['WBBS', 'Thriller', 'Beat It']]
3
4 i = 0
5 while i < len(Music_Folder):
6   j = 0
7   while j < len(Music_Folder[i]):
8     print("Playing",Music_Folder[i][j])
9     j+=1
10  i+=1
```

```
Playing D.D.
Playing Outside
Playing WBBS
Playing Thriller
Playing Beat It
```

> Syntax of nested for-while loop :
>
> ```
> for variable in sequence:
>     statements/optional/
>     initialization
>
>     while condition:
>         statements
>         Increment/Decrement
> ```

Ex 10.8 (Nested for-while Loops)

```
Music_Folder = [['D.D.', 'Outside'],
                ['WBBS', 'Thriller', 'Beat It']]

for i in Music_Folder:
  j = 0
  while j<len(Music_Folder[Music_Folder.index(i)]):
    print("Playing",Music_Folder[Music_Folder.index(i)][j])
    j+=1
```

```
Playing D.D.
Playing Outside
Playing WBBS
Playing Thriller
Playing Beat It
```

> Syntax of nested for-while loop :
>
> `Initialization`
>
> `while condition:`
>
> `    statements/optional/`
>
> `    for variable in sequence:`
>
> `        statements`
>
> `    Increment/Decrement`

Ex 10.9 (Nested while-for Loops)

```
Music_Folder = [['D.D.', 'Outside'],
                ['WBBS', 'Thriller', 'Beat It']]

i = 0
while i < len(Music_Folder):
  for j in Music_Folder[i]:
    print("Playing",Music_Folder[i]
    [Music_Folder[i].index(j)])
  i+=1
```

```
Playing D.D.
Playing Outside
Playing WBBS
Playing Thriller
Playing Beat It
```

EXERCISE 10.6

Find sum of all prime numbers from 0 to 100?

```
# write your answer code here

```

Lesson Four : break, continue, pass Statements

You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution.

Python provides break and continue statements to handle such situations and to have good control on your loop.

break Statement

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

> Syntax of break statement:
> `for variable in sequence:`
> `    statements`
> `    break`

💡 *Suppose you want to play only the first two songs of the album?*

Ex 10.10 (break Statement)

```
Album = ['WBSS', 'Thriller', 'Beat It', 'PYT']

i = 0
while i < len(Album):
  if i == 2:
    break
  print("Playing",Album[i])
  i+=1
```

```
Playing WBSS
Playing Thriller
```

EXERCISE 10.7

Print all vowels in a given string from user?

```
# write your answer code here

```

💡 *Suppose you want to skip 'Thriller' song?*

continue Statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

> Syntax of continue statement:
> `for variable in sequence:`
> `    statements`
> `    continue`

Ex 10.11 (continue Statement)

```
Album = ['WBSS', 'Thriller', 'Beat It', 'PYT']

for i in Album:
  if i == 'Thriller':
    continue
  print("Playing",i)
```

```
Playing WBSS
Playing Beat It
Playing PYT
```

EXERCISE 10.8

Print all prime numbers from 1 to 100 except numbers divisible by 3?

```
# write your answer code here

```

💡 *What if you can't decide what to do with 'Thriller' Song?*

pass Statement

The pass statement is used when you have to include some statement syntactically but doesn't want to perform any operation.

> Syntax of pass statement:
>
> ```
> for variable in sequence:
>     statements
>     pass
> ```

Ex 10.12 (pass Statement)

```
Album = ['WBSS', 'Thriller', 'Beat It', 'PYT']

for i in Album:
  if i == 'Thriller':
    pass
  else:
    print("Playing",i)
```

```
Playing WBSS
Playing Beat It
Playing PYT
```

Difference between pass and comment

In Python programming, the pass statement is a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when the pass is executed. It results in no operation (NOP).

EXERCISE 10.9

Print all even digits in a number from user?

```
# write your answer code here

```

EXERCISE 10.10

Print all vowels in a given string from user?

```
# write your answer code here

```

Lesson Five : List Comphrension

List Cloning using append() method

List cloning refers to creating a new list with the same elements as an existing list. you can create an empty list and then append each item from the original list to the new list.

Ex 10.13 (List Cloning using append() method)

```
original_list = [1, 2, 3, 4, 5]
cloned_list = []

for i in original_list:
    cloned_list.append(i)

print("Original List:", original_list)
print("Cloned List:", cloned_list)
```

```
Original List: [1, 2, 3, 4, 5]
Cloned List: [1, 2, 3, 4, 5]
```

The above code can be written in a single line using list comprehensions.

What is List Comprehension?

List comprehensions are the precise way of creating a list using iterable objects like tuples, strings, lists etc.

> Syntax for list comprehension:
> `list_name = [operation for item in iterable_object]`

Here Iterable represents a list, set, tuple, dictionary or range object. The result of list comprehension is a new list based on the applying conditions.

Ex 10.14 (List Cloning using List Comprehension)

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

cloned_list = [i for i in original_list]

print("Original List:", original_list)
print("Cloned List:", cloned_list)
```

```
Original List: [1, 2, 3, 4, 5]
Cloned List: [1, 2, 3, 4, 5]
```

💡 *What if we want to extrat even numbers from a list of numbers?*

Ex 10.15 (List Comprehensions with if statement)

```
numbers = [1, 4, 7, 10, 13, 16, 19]
print("Original List:", numbers)

list_comp = [i for i in numbers if i%2==0]
print("Compressed List:", list_comp)
```

```
Original List: [1, 4, 7, 10, 13, 16, 19]
Compressed List: [4, 10, 16]
```

Tuple Comprehension

PS. Tuple comprehension is not supported by Python.

When attempting to perform tuple comprehension, the result is not a tuple, it is a list.

Ex 10.16 (Tuple comprehension return a List)

```
numbers = (1, 2, 3, 4, 5)
print("Original Tuple:", numbers)
print(type(numbers))

tuple_comp = [i*2 for i in numbers]
print("Compressed Tuple:", tuple_comp)
print(type(tuple_comp))
```

```
Original Tuple: (1, 2, 3, 4, 5)
<class 'tuple'>
Compressed Tuple: [2, 4, 6, 8, 10]
<class 'list'>
```

However, you can use generator expressions to create tuples in a similar way. A generator expression is enclosed in parentheses instead of square brackets, which are used for list comprehension. The expression after execution will return a generator object rather than a tuple object, so we need to convert it into a tuple.

> Syntax for Tuple comprehension with generator expression:
> `tuple_name = tuple(operation for item in iterable_object)`

Ex 10.17 (Tuple comprehension with generator expression)

```
numbers = (1, 2, 3, 4, 5)
print("Original Tuple:", numbers)
print(type(numbers))

tuple_comp = tuple(i*2 for i in numbers)
print("Compressed Tuple:", tuple_comp)
print(type(tuple_comp))
```

```
Original Tuple: (1, 2, 3, 4, 5)
<class 'tuple'>
Compressed Tuple: (2, 4, 6, 8, 10)
<class 'tuple'>
```

EXERCISE 10.11

Create a list using list comprehension that includes only the prime numbers from the original list?

```
# write your answer code here

```

Lesson Six : Drawing Star (*) Patterns

Ex 10.18 (Draw square pattern)

```
r = int(input("Enter No. of rows:"))

for i in range(1, r + 1):
  for j in range(1, r + 1):
    print('*', end=' ')
  print()
```

```
Enter No. of rows:4
* * * *
* * * *
* * * *
* * * *
```

Ex 10.19 (Draw Right Angle Triangle /Pyramid/)

```
n = int(input("Enter No. of rows:"))

for i in range(0, n):
  for j in range(0, i + 1):
    print("* ", end="")
  print()
```

```
Enter No. of rows:4
*
* *
* * *
* * * *
```

EXERCISE 10.12

Draw an Isosceles Triangle /Christmas Tree/ pattern which looks like as follows.

Enter the number of rows: 4

```
   *
  * *
 * * *
* * * *
```

```
1 # write your answer code here
2
```

Ex 10.20 (Draw Diamond pattern)

```
1 # Upward triangle Pyramid
2 r = int(input("Enter No. of rows:"))
3 k = 2 * r - 2
4
5 for i in range(0, r):
6   for j in range(0, k):
7     print(end=" ")
8   k = k - 1
9   for j in range(0, i + 1):
10    print("* ", end="")
11  print("")
12
13 # Downward triangle Pyramid
14 k = r - 2
15
16 for i in range(r, -1, -1):
17  for j in range(k, 0, -1):
18    print(end=" ")
19  k = k + 1
20  for j in range(0, i + 1):
21    print("* ", end="")
22  print("")
```

```
Enter No. of rows:4
      *
     * *
    * * *
   * * * *
  * * * * *
   * * * *
    * * *
     * *
      *
```

EXERCISE 10.13

Draw Life Span /Hour Glass/ pattern which looks like as follows

Enter the number of rows: 4

```
* * * *
 * * *
  * *
   *
  * *
 * * *
* * * *
```

```
# write your answer code here

```

Lesson Seven : Drawing Number and Alphabet Patterns

Drawing Number Patterns

Ex 10.21 (Draw Numerical Pyramid pattern)

```
r = int(input("Enter No. of rows:"))

for i in range(r + 1):
  for j in range(i):
    print(i, end=" ")
  print(" ")
```

```
Enter No. of rows:4

1
2 2
3 3 3
4 4 4 4
```

Ex 10.22 (Draw Numerical Pyramid with increasing no.)

```
current_NO = 1
stop = 2
r = int(input("Enter No. of rows:"))

for i in range(r):
  for j in range(1, stop):
    print(current_NO, end=' ')
    current_NO += 1
  print("")
  stop += 2
```

```
Enter No. of rows:4
1
2 3 4
5 6 7 8 9
10 11 12 13 14 15 16
```

EXERCISE 10.14

Build a 6 by 6 multiplication table which looks like as follows

Enter the number of rows: 6

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36

```
# write your answer code here

```

Drawing Alphabet Patterns

Ex 10.23 (Draw Alphabetical Pyramid pattern)

```
r = int(input("Enter No. of rows:"))
AV = 65

for i in range(0, r):
  for j in range(0, i + 1):
    alphabet = chr(AV)
    print(alphabet, end=' ')
    AV += 1
  print()
```

```
Enter No. of rows:4
A
B C
D E F
G H I J
```

EXERCISE 10.15

Draw an Alphabetical Triangle pattern which looks like as follows.

Enter the number of rows: 4

```
   A
  B C
 D E F
G H I J
```

```
1 # write your answer code here
2
```

Key Takeaways

- *A loop is a repetitive construction in programming.*
- *Loops provide code reusability and allow traversal over sequence data types.*
- *Python has while loops and for loops.*
- *An infinite loop is a loop that never ends because its condition never becomes false.*
- *A nested loop is a loop that is written inside another loop.*
- *The break statement terminates a loop, the continue statement skips the rest of the code for the current iteration, and the pass statement is used as a placeholder.*
- *Comments are ignored by the interpreter, while the pass statement is not ignored.*
- *List comprehension is a concise way of creating lists using iterable objects like tuples, strings, and lists.*

// END OF CHAPTER TEN

Click here for Chapter Eleven

CHAPTER ELEVEN

Set

Objectives

- *What is a set?*
- *How can you insert elements into a set?*
- *What is the difference between the add() and update() methods?*
- *How can you delete elements from a set or delete the set itself?*
- *What operations can you perform on a set?*
- *What methods can you use on a set?*
- *What is a frozen set?*

Lesson One : Introduction to Set

What is Set?

Sets are also one of the data types just as lists and tuples. If we want to represent a group of unique elements then we can go for sets.

PS. Set cannot store duplicate elements.

Points to remember about set data structure -

- **Insertion order is not preserved.**
- Indexing and slicing are **not allowed.**
- **Nested set** is not allowed.
- Sets can **store the same and different types** of elements or objects.
- Set objects are **mutable** which means once we create a set element we can perform any changes in it.

How To Create a Set?

There are three ways to create a set in Python. Let's see them with examples.

- Using curly brace {}
- Using range() function
- Using set() method

1. Creating a set using curly braces { }

We can create set by using curly braces {} and all elements separated by comma separator in set.

> Syntax :
> `set_name = {elements}`

PS. Remember, [ ] is for lists, ( ) is for tuples and { } is for sets.

💡 *Suppose, Let us create a set of FCB (Barcelona) Forward players.*

Ex 11.1 (Creating a set using curly brace {})

```
1 forward_players = {'Messi', 'Fati', 'Pedri'}
2
3 print("FCB Forwards",forward_players)
4 print(type(forward_players))
```

```
FCB Forwards {'Pedri', 'Fati', 'Messi'}
<class 'set'>
```

2. Creating a set using range() function

As we discussed earlier in creating a list using range function, The range() function creates a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

> Syntax :
> `set_name = set(range(start, stop, step))`

Ex 11.2 (Creating a set using range() function)

```
1 Multiples_of_5 = set(range(5, 51, 5))
2
3 print(Multiples_of_5)
4 print(type(Multiples_of_5))
```

```
{35, 5, 40, 10, 45, 15, 50, 20, 25, 30}
<class 'set'>
```

From the above programs, we can observe that the order of the elements is not preserved.

Ex 11.3 (Duplicates are not allowed)

```
forward_players = {'Messi', 'Fati', 'Pedri', 'Fati'}

print("FCB Forwards",forward_players)
print(type(forward_players))
```

```
FCB Forwards {'Pedri', 'Fati', 'Messi'}
<class 'set'>
```

PS. Duplicates not allowed in python sets.

3. Creating an empty set using set() function

You can create set by using set() function. Generally, we use this function for creating sets from other data types like list, tuple etc…

> Syntax :
> `set_name = list(elements)`

PS. We already discussed earlier in Data Type Chapter Lesson 4 - Data Type Conversion.

Special case ~ empty set

We might think that creating an empty set is just to use empty curly braces {}. But, in python, we have another data type called dictionaries for which we use {} braces only. Hence creating a data type with empty curly braces will be considered as dictionary type rather than set. Then how to create an empty set? We can do so by using the set() function as shown in the below example.

> Syntax :
> `set_name = set()`

Ex 11.4 (Creating an empty set)

```
d = {}
print(d)
print(type(d))

s = set()
print(s)
print(type(s))
```

```
{}
<class 'dict'>
set()
<class 'set'>
```

Lesson Two : Insertion of Set Elements

💡 *Suppose, FCB bought extra forward player, then How to add him to the set?*

1. add() method

The add() method is used to add elements to the set.

> Syntax :
> `set_name.add(new_element)`

Ex 11.5 (add() function)

```
1 forward_players = {'Messi', 'Fati', 'Pedri'}
2
3 forward_players.add('Ronaldo')
4 print("FCB Forwards",forward_players)
```

```
FCB Forwards {'Ronaldo', 'Pedri', 'Fati', 'Messi'}
```

💡 *Suppose, FCB bought extra forward (more than one) players, then How to add them to the set?*

2. update() method

To add multiple items to the set. The arguments are not individual elements, but are sequential data type. like list, tuple, range etc.

> Syntax :
> `set_1.update(set_2)`

Ex 11.6 (update() method)

```
1 forward_players = {'Messi','Fati','Pedri'}
2
3 new_forwards = {'Ronaldo','Neymar'}
4 forward_players.update(new_forwards)
5 print("FCB Forwards\n",forward_players)
```

```
FCB Forwards
 {'Fati', 'Neymar', 'Ronaldo', 'Pedri', 'Messi'}
```

Difference between add() and update()

We can use add() to add individual items to the set, whereas update() is used to add multiple items to set.

The add() method can take one argument whereas update() method can take any number of arguments but the only point is all of them should be sequence data type.

3. copy() method

This method returns a shallow copy of the set.

> Syntax :
> `new_set = copied_set.copy()`

Ex 11.7 (copy() method)

```
forward_players = {'Messi', 'Fati', 'Pedri'}

copy_forwards = forward_players.copy()
print("Copied Forwards",copy_forwards)
```

```
Copied Forwards {'Pedri', 'Fati', 'Messi'}
```

`EXERCISE 11.1`

`Create set of files ~ ( Img_01.png, Avatar.mp4 ), add new file ‘Track01.mp3’ then Rename ‘Track01.mp3’ to ‘Music01.mp3’, finally Copy all files to new set (Folder)?`

```
# write your answer code here

```

Lesson Three : Deletion of Set Elements

💡 *Suppose, The club FCB is in bankruptcy want to sell one random forward player. How could you remove him from the set? Let us see…*

1. pop() method

This method removes an arbitrary / random element from the set and returns the element removed.

> Syntax :
> `set_name.pop()`

Ex 11.8 (pop() method)

```
forward_players = {'Messi', 'Fati', 'Pedri'}

print(forward_players.pop(),"is sold.")
print("FCB Forwards",forward_players)
```

```
Messi is sold.
FCB Forwards set()
```

💡 *Hell No… Messi is crucial player for FCB so let us keep him and sell another player ~ Pedri*

2. remove() method

This method removes specific elements from the set. If the specified element is not present in the set then we will get KeyError.

> Syntax :
> `set_name.remove(element)`

Ex 11.9 (remove() method)

```
forward_players = {'Messi', 'Fati', 'Pedri'}

forward_players.remove('Pedri')
print("FCB Forwards",forward_players)
```

```
FCB Forwards {'Fati', 'Messi'}
```

Ex 11.10 (remove() KeyError)

```
forward_players = {'Messi', 'Fati', 'Pedri'}

forward_players.remove('Ronaldo')
print("FCB Forwards",forward_players)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-28-ddb392e3944f> in <cell line: 3>()
      1 forward_players = {'Messi', 'Fati', 'Pedri'}
      2
----> 3 forward_players.remove('Ronaldo')
      4 print("FCB Forwards",forward_players)

KeyError: 'Ronaldo'
```

3. discard() method

This method removes the specified element from the set. If the specified element is not present in the set, then **we won't get any error.**

> Syntax :
> `set_name.discard(element)`

Ex 11.11 (discard() method)

```
forward_players = {'Messi', 'Fati', 'Pedri'}

forward_players.discard('Pedri')
print("FCB Forwards",forward_players)
```

```
FCB Forwards {'Fati', 'Messi'}
```

Ex 11.12 (discard() No KeyError)

```
forward_players = {'Messi', 'Fati', 'Pedri'}

forward_players.discard('Ronaldo')
print("FCB Forwards",forward_players)
```

```
FCB Forwards {'Pedri', 'Fati', 'Messi'}
```

4. clear() method

This method removes all elements from the set.

> Syntax :
> `set_name.clear()`

Ex 11.13 (clear() method)

```
forward_players = {'Messi', 'Fati', 'Pedri'}

forward_players.clear()
print("FCB Forwards",forward_players)
```

```
FCB Forwards set()
```

`EXCERCISE 11.2`

`Create set of files ~ ( Img_01.png, Pic.jpeg, Avatar.mp4 ), then delete all image files?`

```
# write your answer code here

```

Lesson Four: Mathematical Operations on Sets

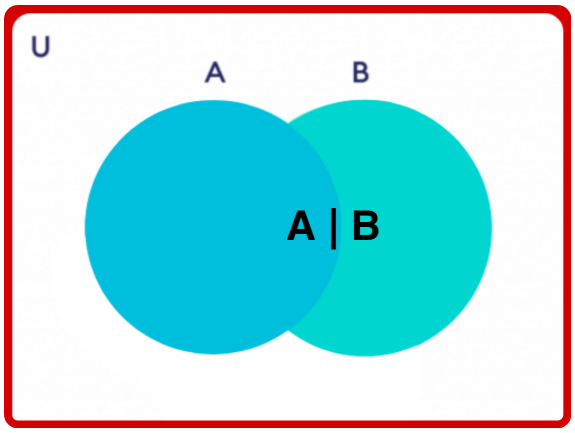
💡 *Suppose, you want to know all Spanish and FCB Forward players? How could you do it? Let us see…*

1. union() method

This method return all elements present in both sets.

> Syntax:
> `A.union(B) or A | B`

PS. Union is commutative, A | B = B | A



Ex 11.14 (union() method)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri', 'Torres'}

print("Spanish & FCB Players:",barcelona_forwards | spain_forwards)
```

```
Spanish & FCB Players: {'Torres', 'Fati', 'Pedri', 'Messi'}
```

EXCERCISE 11.3

Create two sets Folder_1 and Folder_2, then print all files in Folder_1 and Folder_2?

```
# write your answer code here

```

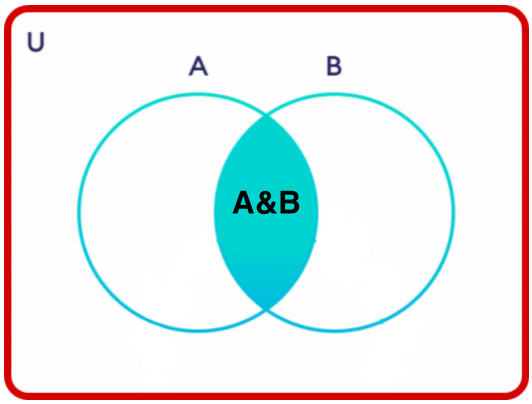
💡 *Suppose, you want to know all FCB Spanish Forward players? How could you do it? Let us see…*

2. intersection() method

This method returns common elements present in both set A and set B.

> Syntax :
> `A.intersection(B) or A & B`

PS. Intersection is commutative, A & B = B & A



Ex 11.15 (intersection() method)

```
1 barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
2 spain_forwards = {'Fati', 'Pedri', 'Torres'}
3
4 print("Spanish FCB Players:",spain_forwards & barcelona_forwards)
```

```
Spanish FCB Players: {'Pedri', 'Fati'}
```

3. intersection_update()

The intersection_update() updates the set calling intersection_update() method with the intersection of sets.

> Syntax :
> `A.intersection_update(B)`

Ex 11.16 (intersection_update() method)

```
1 barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
2 spain_forwards = {'Fati', 'Pedri', 'Torres'}
3
4 barcelona_forwards.intersection_update(spain_forwards)
5 print("Spanish FCB Players:",barcelona_forwards)
```

```
Spanish FCB Players: {'Pedri', 'Fati'}
```

EXERCISE 11.4

Create two sets Folder_1 and Folder_2, then delete all duplicated files?

```
# write your answer code here

```

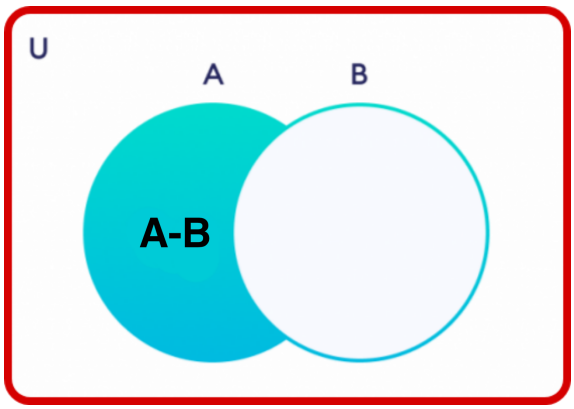
💡 *Suppose, you want to know all Non-Spanish FCB Forward Players? or Spanish but not FCB Forward Players? How could you do it? Let us see…*

4. difference() method

This method returns the elements present in set A but not in set B.

> Syntax :
> `A.difference(B) or A - B`

PS. difference is not commutative, A-B != B-A



Ex 11.17 (difference() method)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri', 'Torres'}

print("Non-Spanish FCB Player:",barcelona_forwards - spain_forwards)
print("Spanish, not FCB Player:",spain_forwards - barcelona_forwards)
```

```
Non-Spanish FCB Player: {'Messi'}
Spanish, not FCB Player: {'Torres'}
```

5. difference_update() method

The difference_update() updates the set calling difference_update() method with difference of sets.

> Syntax :
> `A.difference_update(B)`

Ex 11.18 (difference_update() method)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri', 'Torres'}

barcelona_forwards.difference_update(spain_forwards)
print("Non-Spanish FCB Player:",barcelona_forwards)
```

```
Non-Spanish FCB Player: {'Messi'}
```

Ex 11.19 (difference_update() method)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri', 'Torres'}

spain_forwards.difference_update(barcelona_forwards)
print("Spanish, not FCB Player:",spain_forwards)
```

```
Spanish, not FCB Player: {'Torres'}
```

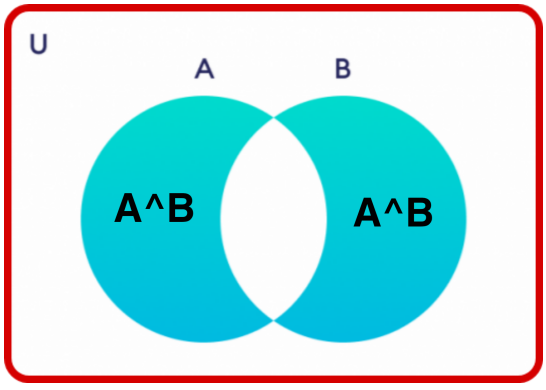
💡 *What if you want to know all Forward Players who plays for FCB or Spain, but not both?*

6. symmetric_difference()

This method returns elements present in either set A or set B but not in both.

> Syntax :
> `A.symmetric_difference(B) or A ^ B`

PS. symmetric_difference is commutative, A ^ B = B ^ A



Ex 11.20 (symmetric_difference() method)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri', 'Torres'}

print("Not played for both FCB & Spain")
print(barcelona_forwards ^ spain_forwards)
```

```
Not played for both FCB & Spain
{'Messi', 'Torres'}
```

7. symmetric_difference_update()

The symmetric_difference_update() method finds the symmetric difference of two sets and updates the set calling it.

> Syntax:
> `A.symmetric_difference_update(B)`

Ex 11.21 (symmetric_difference_update() method)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri', 'Torres'}

barcelona_forwards.symmetric_difference_update(spain_forwards)
print("Not play for both FCB & Spain")
print(barcelona_forwards)
```

```
Not play for both FCB & Spain
{'Messi', 'Torres'}
```

`EXERCISE 11.5`

`Create two sets Folder_1 and Folder_2, then print all non-duplicated files?`

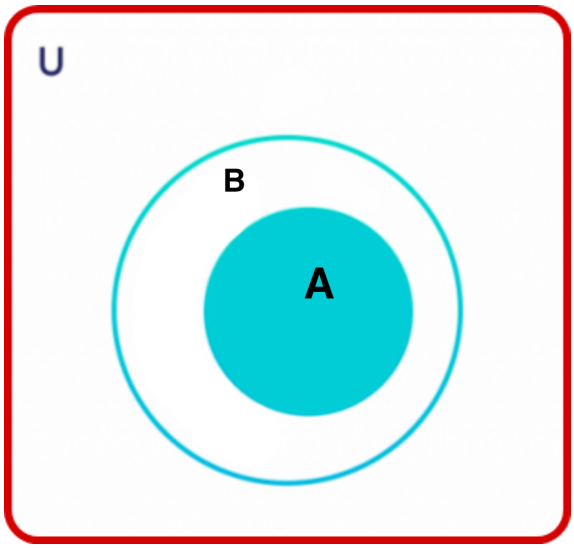
```
# write your answer code here

```

8. issubset(), issuperset() and isdisjoint()

issubset() method returns True if all elements of a set are present in another set. If not, it returns False.

> Syntax :
> `A.issubset(B)`



Ex 11.22 (membership operators in sets)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri'}

print(barcelona_forwards.issubset(spain_forwards))
print(spain_forwards.issubset(barcelona_forwards))
```

```
False
True
```

issuperset() method returns True if a set has every elements of another set. If not, it returns False.

> Syntax :
>
> `A.issuperset(B)`

Ex 11.23 (membership operators in sets)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri'}

print(barcelona_forwards.issuperset(spain_forwards))
print(spain_forwards.issuperset(barcelona_forwards))
```

```
True
False
```

isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False.

> Syntax :
>
> `A.isdisjoint(B)`

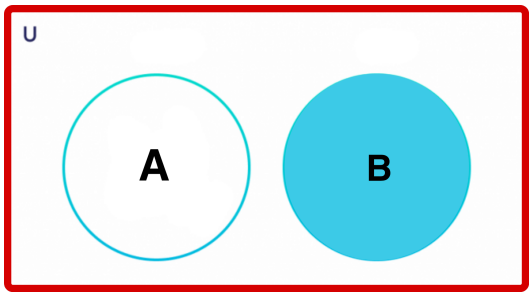
PS. All issubset(), issuperset() and isdisjoint() methods are not commutative.

Ex 11.24 (isdisjoint() method)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}
spain_forwards = {'Fati', 'Pedri'}
argentinian_forwards = {'Messi','Aguero'}

print(barcelona_forwards.isdisjoint(spain_forwards))
print(spain_forwards.isdisjoint(argentinian_forwards))
print(argentinian_forwards.isdisjoint(spain_forwards))
```

```
False
True
True
```



EXERCISE 11.6

Create two sets Folder_1 & Folder_2, then delete duplicated (subset) folder and keep super folder?

```
# write your answer code here

```

Frozen Set

Frozen set is similar to a set, but it is immutable, meaning its elements cannot be changed after creation.

> Syntax :
> `frozen_set_name = frozenset(set_name)`

Ex 11.25 (Creating a frozen set)

```
barcelona_forwards = {'Messi', 'Fati', 'Pedri'}

frozen_barcelona_forwards = frozenset(barcelona_forwards)
print(frozen_barcelona_forwards)
print(type(frozen_barcelona_forwards))
```

```
frozenset({'Messi', 'Pedri', 'Fati'})
<class 'frozenset'>
```

`EXERCISE 11.7`

`Create a frozen set of files / Zip File /?`

```
# write your answer code here

```

Other Mathematical Operations

The following methods also work for set as same as list

- len()
- count()
- max(), min(), sum()
- multiplication *
- Relational Operator
- Membership operator

Key Takeaways

- *A set is a collection of unique elements.*
- *You can insert elements into a set using the* `add(), update()` *and* `copy()` *methods.*
- *The* `add()` *method is used to add individual items to a set, while the* `update()` *method is used to add multiple items to a set.*
- *You can delete elements from a set using the* `remove(), pop(), discard()` *and* `clear()` *methods.*
- *You can perform operations such as* `len(), count(), max(), min(), sum()` *multiplication, membership and relational operators on a set.*
- *You can use methods like* `union(), intersection(), intersection_update(), difference(), difference_update(), symmetric_difference(), symmetric_difference_update(), issubset(), issuperset()` *and* `isdisjoint()` *methods on a set.*
- *A frozen set is similar to a set, but it is immutable, meaning its elements cannot be changed after creation.*

// END OF CHAPTER ELEVEN

Click here for Chapter Twelve

CHAPTER TWELVE

Dictionary

Objectives

- *What is a dictionary?*
- *How can you insert and update elements in a dictionary?*
- *How can you access dictionary elements?*
- *How can you delete elements from a dictionary or delete the dictionary itself?*
- *What is dictionary comprehension?*

Lesson One : Introduction to Dictionary

In python, to represent a group of individual objects as a single entity then we have used lists, tuples, sets. If we want to represent a group of objects as keyvalue pairs then we should go for dictionaries.

What is Dictionary?

Dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.

Characteristics of Dictionary

- Dictionary will contain data in the form of **key, value pairs**.
- Key and values are **separated by colon :** symbol.
- One key – value pair can be represented as an item.
- Duplicate **keys are not allowed** but Duplicate **values can be allowed**.
- **Heterogeneous objects** are allowed for both key and values.
- Insertion order is **not preserved**.
- Dictionary objects are **mutable** & **dynamic**.
- **Indexing and slicing** concepts are **not applicable**.

How to Create dictionary?

There are multiple ways to create a dictionary.

- Using curly brace {}
- Using dict() function
- Using curly brace {} and colon (:)
- Using fromkeys() method
- Using zip() function

1. Using curly brace { }

We can create empty dictionaries by using curly braces. And later on we can add the key value pairs into it.

> Syntax :
> `dictionary_name = {}`

Ex 12.1 (creating an empty dictionary)

```
1 p = {}
2
3 print(p)
4 print(type(p))
```

```
{}
<class 'dict'>
```

2. Using dict() function

This can be used to create an empty dictionary. Later, we can add elements to that created dictionary.

> Syntax :
> `dictionary_name = dict()`

Ex 12.2 (Creating dictionary using dict() function)

```
1 p = dict()
2
3 print(p)
4 print(type(p))
```

```
{}
<class 'dict'>
```

3. Using curly brace { } and colon (:)

It creates dictionary with specified elements.

> Syntax:
> `dictionary_name = {key1:value1,..., keyN:valueN}`

💡 *Suppose, you want to create dictionary of English Premier League Top Goal Scorers with respect to their total goal. How could you do it? Let us see…*

Ex 12.3 (creating dictionary using { } & colon)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print(goal_scorers)
print(type(goal_scorers))
```

```
{'Salah': 13, 'Kane': 12, 'Son': 12}
<class 'dict'>
```

💡 *What if, All 3 players scored equal no. of goals?*

4. Using fromkeys() method

The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.

> Syntax :
> `dictionary_name.fromkeys(A, value)`

PS. A can be any sequence data type.

Ex 12.4 (creating dictionary using fromkeys())

```
players = {'Salah', 'Kane', 'Son'}
goals = 13

goal_scorers = dict.fromkeys(players, goals)
print(goal_scorers)
print(type(goal_scorers))
```

```
{'Kane': 13, 'Son': 13, 'Salah': 13}
<class 'dict'>
```

PS. value is optional, default = None.

Ex 12.5 (creating dictionary using fromkeys())

```
players = {'Salah', 'Kane', 'Son'}
goals = 13

goal_scorers = dict.fromkeys(players)
print(goal_scorers)
print(type(goal_scorers))
```

```
{'Kane': None, 'Son': None, 'Salah': None}
<class 'dict'>
```

5. Using zip() function

The zip() function takes one or more sequences and combines the corresponding items in the sequences into a dictionary. It stops when the shortest sequence is exhausted.

> Syntax:
> `dictionary_name = dict(zip(A, B))`

PS. A and B can be any sequence data type.

Ex 12.6 (Creating dictionary using zip() function)

```
players = ('Salah', 'Kane', 'Son')
goals = [13, 12, 12]

goal_scorers = dict(zip(players, goals))
print(goal_scorers)
print(type(goal_scorers))
```

```
{'Salah': 13, 'Kane': 12, 'Son': 12}
<class 'dict'>
```

💡 *Suppose, A player ~ "Vardy" scored more goals and he becomes Top Goal Scorer. How to add Vardy to the dictionary? Let us see…*

Lesson Two : Insertion of Dictionary Elements

How to insert elements to a dictionary?

There is no explicitly defined method to add a new key to the dictionary. If you want to add a new key to the dictionary, then you can use assignment operator with dictionary key.

> Syntax:
>
> `dictionary_name[key] = value`

Ex 12.7 (Inserting elements to a dictionary)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Old Top Goal Scorers\n',goal_scorers)
goal_scorers['Vardy'] = 11
print('New Top Goal Scorers\n',goal_scorers)
```

```
Old Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12}
New Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12, 'Vardy': 11}
```

💡 *Suppose, "Son" scored two more goals in last night game. How to update Son's goals? Let us see…*

How to Update dictionary elements?

We can also update the value for a particular existing key in a dictionary.

> Syntax:
>
> `dictionary_name[key] = value`

Ex 12.8 (Updating a dictionary element)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Old Top Goal Scorers\n',goal_scorers)
goal_scorers['Son'] = 14
print('New Top Goal Scorers\n',goal_scorers)
```

```
Old Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12}
New Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 14}
```

update() method

The update() method updates the dictionary with the elements from the another dictionary object or from an iterable of key/value pairs.

> Syntax:
>
> `dictionary_name.update(key=value,...)`

Ex 12.9 (update() method case 1)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

goal_scorers.update(Haaland=11, Vardy=12, Son=14)
print(goal_scorers)
```

```
{'Salah': 13, 'Kane': 12, 'Son': 14, 'Haaland': 11, 'Vardy': 12}
```

PS. element can be another dictionary.

> Syntax:
> `dictionary_name.update(new_dictionary)`

Ex 12.10 (update() method case 2)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}
new_scorers = {'Haaland':11, 'Vardy':12, 'Son':14}

goal_scorers.update(new_scorers)
print(goal_scorers)
```

```
{'Salah': 13, 'Kane': 12, 'Son': 14, 'Haaland': 11, 'Vardy': 12}
```

`EXERCISE 12.1`

`Create a contact dictionary which contains phone no. and name as key-value pair then insert a contact Abel, +251 9 12546782, finally update its phone no. to +251 9 23468494?`

```
# write your answer code here

```

Lesson Three : Accessing a Dictionary

How to access a Dictionary?

There are multiple ways in which you can access the elements in the dictionary. They are:

- By using key
- By using get() method
- By using setdefault() method
- By using random(choice()) method
- By using items(), keys(), values() methods

💡 *What if you want to know how many goals did "Son" scored in Premier League?*

1. Using key

We can access dictionary values by using keys. Keys play a main role to access the data.

> Syntax :
> `dictionary_name[key]`

Ex 12.11 (Accessing dictionary values using keys)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print("Son's Goals:",goal_scorers['Son'])
```

```
Son's Goals: 12
```

PS. While accessing, if specified key is not available then we will get KeyError.

Ex 12.12 (KeyError)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print("Messi's Goals:",goal_scorers['Messi'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-104a187712c7> in <cell line: 3>()
      1 goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}
      2
----> 3 print("Messi's Goals:",goal_scorers['Messi'])

KeyError: 'Messi'
```

💡 *How to solve this KeyError? Let us see…*

2. get() method

This method used to get the value associated with the key. This is another way to get the values of the dictionary based on the key. The biggest advantage it gives over the normal way of accessing a dictionary is, this doesn't give any error if the key is not present.

> Syntax:
> `dictionary_name.get(key)`

- **Case 1:** If the key is available, then it returns the corresponding value otherwise returns None. It won't raise any error.
- **Case 2:** If the key is available, then returns the corresponding value otherwise returns default value which we give.

> Syntax :
> `dictionary_name.get(key, default value)`

Ex 12.13 (get() method case 1)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print("Son's Goal:",goal_scorers.get('Son'))
print("Messi's Goal:",goal_scorers.get('Messi'))
```

```
Son's Goal: 12
Messi's Goal: None
```

Ex 12.14 (get() method case 2)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print("Son's Goal:",goal_scorers.get('Son'))
print(goal_scorers.get('Messi',"Messi isn't a premier league player."))
```

```
Son's Goal: 12
Messi isn't a premier league player.
```

How to handle this KeyError?

We can handle this error by checking whether a key is already available or not by using in operator.

PS. There is also another precise way of handling this error, which we shall discuss later in Exception Handling Chapter.

Ex 12.15 (Handle KeyError in Python Dictionary)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

if 'Messi' in goal_scorers:
  print("Messi's Goal:")
  print(goal_scorers['Messi'])
else:
  print("Messi isn't found here!")
```

```
Messi isn't found here!
```

EXERCISE 12.2

Create a contact dictionary which contains phone no. and name as key-value pair then search a contact Phone no. of 'Abel'?

```
# write your answer code here

```

💡 *What about other Goal Keepers?*

3. setdefault() method

The setdefault() method returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.

> Syntax:
> `dictionary_name.setdefalut(key, value)`

Ex 12.16 (setdefault() method)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Son Goals:',goal_scorers.setdefault('Son'))
print('Alison Goals:',goal_scorers.setdefault('Alison'))
print(goal_scorers)
```

```
Son Goals: 12
Alison Goals: None
{'Salah': 13, 'Kane': 12, 'Son': 12, 'Alison': None}
```

PS. value is optional, default = None.

Ex 12.17 (setdefault() method)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Son Goals:',goal_scorers.setdefault('Son',0))
print('Alison Goals:',goal_scorers.setdefault('Can',0))
```

```
Son Goals: 12
Alison Goals: 0
```

EXERCISE 12.3

Create a contact dictionary which contains phone no and name then set default Name as 'Unknown'?

```
# write your answer code here

```

💡 *Suppose, Premier League completed, but there are three top goal scorers, who have scored equal No. of goals = 23. Which player should be awarded? How to solve this problem? Let us see…*

4. random.choice() method

Sometimes, while working with dictionaries, we can have a situation in which we need to find a random pair from dictionary. This type of problem can come in games such as lotteries etc.

> Syntax:
> `import random`
> `random.choice(list(dictionary_name.keys()))`

Ex 12.18 (random(choice()) method)

```
import random
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print("The Winner is " ,end="")
print(random.choice(list(goal_scorers.keys())))
```

```
The Winner is Salah
```

EXERCISE 12.4

Create a contact dictionary which contains phone no. and name then call to random contact?

```
# write your answer code here

```

5. items(), keys(), values() methods

A key value pair in a dictionary is called an item.

- **items()** method returns list of tuples representing key-value pairs.

> Syntax:
> `dictionary_name.items()`

- **keys()** method returns all keys.

> Syntax:
> `dictionary_name.keys()`

- **values()** method returns all values.

> Syntax:
> `dictionary_name.values()`

Ex 12.19 (items(), keys(), values() methods)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print(goal_scorers)
print("\nPlayer-Goal Pair\n",goal_scorers.items())
print('\nTop Goal Scorers\n',goal_scorers.keys())
print('\nTop Goals:',goal_scorers.values())
```

```
{'Salah': 13, 'Kane': 12, 'Son': 12}

Player-Goal Pair
 dict_items([('Salah', 13), ('Kane', 12), ('Son', 12)])

Top Goal Scorers
 dict_keys(['Salah', 'Kane', 'Son'])

Top Goals: dict_values([13, 12, 12])
```

How to copy a dictionary?

We use copy() method to create exactly duplicate dictionary (cloned copy)

Ex 12.20 (copy() method)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Premier League Top Goal Scorers:\n',goal_scorers)
FA_Cup_top_scorers = goal_scorers.copy()
print('\nFA Cup Top Goal Scorers:\n',FA_Cup_top_scorers)
```

```
Premier League Top Goal Scorers:
 {'Salah': 13, 'Kane': 12, 'Son': 12}

FA Cup Top Goal Scorers:
 {'Salah': 13, 'Kane': 12, 'Son': 12}
```

EXERCISE 12.5

Create a contact dictionary which contains phone no. and name then print all phone no. and name of contacts?

```
# write your answer code here

```

Lesson Four : Deletion of a Dictionary

There are many ways of deleting & clearing elements from a dictionary. Let's try to understand each.

💡 *Suppose, A player – "Kane" accused for doping, then he disqualified from the league. How to remove him from top goal scorers dictionary? Let us see…*

1. del keyword

del keyword deletes an item of specified key.

> Syntax:
> `del dictionary_name[key]`

Ex 12.21 (del keyword)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Old Top Goal Scorers\n',goal_scorers)
del goal_scorers['Kane']
print('\nNew Top Goal Scorers\n',goal_scorers)
```

```
Old Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12}

New Top Goal Scorers
 {'Salah': 13, 'Son': 12}
```

PS. If key isn't available, then we will get KeyError.

Ex 12.22 (del keyword KeyError)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Old Top Goal Scorers\n',goal_scorers)
del goal_scorers['Messi']
print('New Top Goal Scorers\n',goal_scorers)
```

```
Old Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12}
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-32-97d1417b265b> in <cell line: 4>()
      2
      3 print('Old Top Goal Scorers\n',goal_scorers)
----> 4 del goal_scorers['Messi']
      5 print('New Top Goal Scorers\n',goal_scorers)

KeyError: 'Messi'
```

2. pop() method

This method removes an item of specified key and returns the corresponding value.

> Syntax:
>
> `dictionary_name.pop(key)`

Ex 12.23 (pop() method)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Old Top Goal Scorers\n',goal_scorers)
goal_scorers.pop('Kane')
print('\nNew Top Goal Scorers\n',goal_scorers)
```

```
Old Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12}

New Top Goal Scorers
 {'Salah': 13, 'Son': 12}
```

PS. If the specified key is not available, then we will get KeyError.

Ex 12.24 (pop() method KeyError)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Old Top Goal Scorers\n',goal_scorers)
goal_scorers.pop('Messi')
print('New Top Goal Scorers\n',goal_scorers)
```

```
Old Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12}
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-35-e34871d094c7> in <cell line: 4>()
      2
      3 print('Old Top Goal Scorers\n',goal_scorers)
----> 4 goal_scorers.pop('Messi')
      5 print('New Top Goal Scorers\n',goal_scorers)

KeyError: 'Messi'
```

3. popitem() method

This method removes an arbitrary item(key-value) from the dictionary and returns it.

> Syntax:
>
> `dictionary_name.popitem()`

Ex 12.25 (popitem() method)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('Old Top Goal Scorers\n',goal_scorers)
goal_scorers.popitem()
print('\nNew Top Goal Scorers\n',goal_scorers)
```

```
Old Top Goal Scorers
 {'Salah': 13, 'Kane': 12, 'Son': 12}

New Top Goal Scorers
 {'Salah': 13, 'Kane': 12}
```

PS. If the dictionary is empty, then we will get KeyError.

Ex 12.26 (popitem() method KeyError)

```
p = {}

p.popitem()
print("After popitem:", p)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-38-dde00e1ca7ba> in <cell line: 3>()
      1 p = {}
      2
----> 3 p.popitem()
      4 print("After popitem:", p)

KeyError: 'popitem(): dictionary is empty'
```

4. clear() method

Clear() method removes all entries in the dictionary. After deleting all entries, it just keeps an empty dictionary.

> Syntax:
> `dictionary_name.clear()`

Ex 12.27 (clear() method)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

goal_scorers.clear()
print(goal_scorers)
```

```
{}
```

`EXERCISE 12.6`

`Create a contact dictionary which contains phone no. and name then delete contact of Abel?`

```
# write your answer code here

```

len() function

This function returns the no. of items in a dictionary.

> Syntax:
> `len(dictionary_name)`

Ex 12.28 (len() function)

```
goal_scorers = {'Salah':13, 'Kane':12, 'Son':12}

print('There are',len(goal_scorers),'top goal scorers.')
```

```
There are 3 top goal scorers.
```

Dictionary Comprehension

Just as we had list comprehensions, we also have this concept applicable for dictionaries even.

Dictionary comprehension is a concise way of creating dictionaries in Python using an iterable object, such as a list, tuple, or another dictionary. It allows you to create dictionaries with specific key-value pairs based on a specified condition or transformation.

> Syntax for dictionary comprehension:
> `{key_expression: value_expression for element in iterable if condition}`

Ex 12.29 (dictionary comprehension)

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

squared_numbers = {i: i**2 for i in numbers}
print(squared_numbers)
print(type(squared_numbers))
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
<class 'dict'>
```

EXERCISE 12.7

Create a dictionary from a list of favorite football players, with player names as keys and character counts as values using dictionary comprehension?

Hint:
List -> ["Ronaldo", "Messi", "Neymar"]
Dictionary -> {"Ronaldo": 7, "Messi": 5, "Neymar": 6}

```
# write your answer code here

```

Key Takeaways

- *A dictionary is an unordered collection of items.*
- *You can insert elements into a dictionary using the assignment operator and* `update()` *method.*
- *You can access dictionary elements using keys,* `get()`, `setdefault()`, `random.choice()`, `items()`, `keys()` *and* `values()` *methods.*
- *You can delete elements from a dictionary using* `del` *keyword,* `pop()`, `popitem()` *and* `clear()` *methods.*
- *Dictionary comprehension is a concise way of creating dictionaries in Python using an iterable object.*

// END OF CHAPTER TWELVE

Click here for Chapter Thirteen

CHAPTER THIRTEEN

Object Oriented Programming

Objectives

- *What are different types of programming paradigms?*
- *What is Object-oriented programming?*
- *What is Class and Object?*
- *What is Constructor?*
- *What is self?*
- *What are class variables?*
- *What is method?*
- *What is static variable?*
- *What are class methods and static methods?*
- *What are local variables?*
- *What are setter and getter methods?*
- *What is Encapsulation?*
- *What is Name Mangling?*
- *What is nested class?*
- *What is garbage collection?*
- *What is inheritance?*
- *What are different types of inheritance?*
- *What is Polymorphism?*
- *What is Overriding?*
- *What is super() function in Python?*
- *What is Overloading?*
- *What is Duck Typing?*
- *What is Abstraction?*
- *What is Interface?*

Lesson One : Introduction to Object Oriented Programming

Programming Before Object Oriented Programming (OOPs)

Before OOPs (object oriented programming), there were two approaches, procedure-oriented programming and functional oriented programming, being used.

💡 *What are these?*

All the three approaches mentioned above are programming paradigms. A programming paradigm is nothing but the style of writing a program. We can also call them as methodologies which are used in writing a software program. Different languages follow different methodologies.

Functional Oriented Programming

Functional oriented programming is a program paradigm where programs are built around functions. Functions can be used as parameters, can be returned, can be used in other functions etc. Here functions are everything and they should always return the same output for a particular input.

Procedural Oriented Programming

Procedural programming is like following a step-by-step procedure or a set of instructions. It emphasizes executing instructions in a specific order, manipulating shared data as needed. It's like cooking a meal by following a recipe that tells you what to do at each step.

After OOPs

Object-oriented programming (OOP) is a way of organizing and designing computer programs. It focuses on creating reusable pieces of code called objects, which can be thought of as individual "things" that have specific characteristics and can perform certain actions.

💡 *Where does Python belong to?*

Python follows both functional oriented and object oriented programming approaches.

Concept of Class and Object

💡 *Imagine you have a TV factory. In order to manufacture televisions, you need a blueprint that tells you how to build them. This blueprint is called a* ***class****. It outlines the common features and behaviors that all TVs produced in the factory will have.*

The class for a television might include specifications such as the brand name, model number, and screen size. It would also describe what a television can do, like turning on and off, changing channels, and adjusting the volume.

Based on this blueprint, you can create actual televisions. Each of these televisions is an "object" that follows the instructions or steps provided by the class. The constructor is like a customized set of instructions that are executed automatically when a new TV object is created based on the class.

The constructor could be a step where we input the brand, model, and size of the TV. So, when we create a new TV object using the class, the constructor will be automatically triggered, and the specified values will be assigned to the respective properties of the TV object.

Just like in the factory, where multiple TVs are made using the same blueprint, you can create multiple TV objects using the same class.

Each television object you create has its own unique characteristics. For example, one TV object might be a Samsung brand with a model number "ABC123" and a screen size of 55 inches. Another TV object could be an LG brand with a model number "XYZ789" and a screen size of 65 inches.

Think of the class as the blueprint that guides the creation of TVs, and the objects as the actual televisions produced using that blueprint. The class defines what a television is and what it can do, while the objects represent specific instances of televisions that exist in the real world.

Once you have a television object, you can interact with it. You can turn it on or off, change the channel, or adjust the volume. These actions are like the behaviors defined in the class. Each television object knows how to perform these actions because it was created based on the class's instructions.

What is Class?

A class is like a blueprint or a template for creating objects. It defines the properties (also known as attributes or variables) and behaviors (also known as methods or functions) that objects of that class will have. Think of a class as a description of what an object should look like and what it should be able to do.

What is Object?

An object, on the other hand, is an instance or a specific occurrence of a class. It is created based on the blueprint provided by the class. You can think of an object as a real-world example based on a blueprint.

What is Constructor?

Constructor is a special method within a class that is automatically called when an object of that class is created. It is used to initialize the object's attributes or perform any necessary setup operations. It’s not required to explicitly invoke or call it.

How to define a class?

> Syntax for defining a class:
>
> ```
> class class_name:
>     // variables or properties
>     // functions or methods
> ```

💡 *What is self in the above syntax?*

What is self?

self is a default variable that refers to a current class object.

By using a self variable, we can access instance variables and instance methods. It can also be some other name rather than ‘self’, but the most commonly used and preferred is ‘self’.

Main reasons why we use self

- **Accessing Instance Variables:** When you create an object from a class, it has its own set of variables. By using self, you can access these variables and see or change their values. It's like having access to the details of an individual object.
- **Differentiating between Instance and Local Variables:** Sometimes, a method (a function inside a class) might have variables with the same name as the instance variables. Using self helps you distinguish between these variables and prevents confusion. Without self, you would create new, separate variables that only exist within the method, and they wouldn't affect the object's attributes.
- **Method Invocation:** When you call a method on an object, you need to use self to refer to that specific object.
- **Object Identity:** In a store full of televisions, each TV is unique and has its own identity. Similarly, when you have multiple objects created from the same class, self helps you refer to a particular object and perform actions on it. It ensures that the correct object is being acted upon and maintains its individuality.

Ex 13.1 (Define a class)

```
1 class Television:
2   brand = "Sony"
3 
4   def turn_on(self):
5     print(self.brand, "TV is on.")
```

The above example code doesn't give any output because we haven't created any object to access the class members (attributes).

How to create Objects?

> Syntax for creating object of a class:
> `object_name = class_name()`

Ex 13.2 (Creating an object)

```
my_tv = Television()
print(my_tv)
```

```
<__main__.Television object at 0x79d7ec12ead0>
```

`--str--()` is one of the magical methods. Whenever we are trying to print any object reference internally, then this method will be called. It returns output, about the location of the object, in the string format

> Syntax:
> `--str--() method:`

Ex 13.3 (Using str() method to print a custom object)

```
class Television:
  brand = "Sony"

  def turn_on(self):
    print(self.brand, "TV is on.")

  def __str__(self):
      return (f'This is a {self.brand} Television object')

my_tv = Television()
print(my_tv)
```

```
This is a Sony Television object
```

Ex 13.4 (Data type of an object)

```
type(my_tv)
```

```
__main__.Television
```

After creating the object of a class, we can access the attributes available in that class with the created object as shown below.

- `object_name.method_name()` or `class_name.method_name(object_name)`
- `object_name.variable_name`

Ex 13.5 (Calling method and variable)

```
my_tv.turn_on()             # calling method using object name
Television.turn_on(my_tv)   # calling method using class name
my_tv.brand                 # calling variable
```

```
Sony TV is on.
Sony TV is on.
'Sony'
```

`EXERCISE 13.1`

`Create a class called "Bank" with bank_name property and display_bank() method?`

```
# write your answer code here

```

Lesson Two : Constructor

How to initialize a Constructor?

Generally, the constructor is used for initial intializations that are required during the object creation. In python, constructor is a method with the name '**init**'. The methods first parameter should be 'self' (referring to the current object).

> Syntax:
> `def --init--(self):`
> `    // body of constructor`

💡 *Is constructor mandatory in Python?*

No, it's not mandatory for a class to have a constructor. It completely is based on our requirement whether to have a constructor or not. At the time of object creation if any initialization is required then we should go for a constructor, else it's not needed.

Ex 13.6 (Define a class with Constructor)

```
class Television:
   def __init__ (self):
       print("constructor is executed automatically at the time of TV creation.")

my_tv = Television()
```

```
    constructor is executed automatically at the time of TV creation.
```

💡 *Can a constructor be called explicitly?*

Yes, we can call constructor explicitly with object name. But since the constructor gets executed automatically at the time of object creation, it is not recommended to call it explicitly.

Ex 13.7 (Calling Constructor Explicitly)

```
class Television:
   def __init__ (self):
       print("constructor is executed automatically at the time of TV creation.")

my_tv = Television()
my_tv.__init__()
```

```
    constructor is executed automatically at the time of TV creation
    constructor is executed automatically at the time of TV creation
```

Constructors can be categorized into two types: parameterized and non-parameterized constructors.

1. Non-Parameterized Constructor (Default Constructor):

A non-parameterized constructor is a constructor that does not take any parameters. It is defined without any arguments and sets up the initial state of the object with default values.

Ex 13.8 (Non Parameterized Constructor)

```
class Television:
   def __init__ (self):
       print("constructor is executed automatically at the time of TV creation.")
       print(self)

my_tv = Television()
```

```
    constructor is executed automatically at the time of TV creation.
    <__main__.Television object at 0x7b2607d13970>
```

2. Parameterized Constructor

A parameterized constructor is a constructor that takes one or more parameters as input. It allows you to pass specific values during object creation and use those values to initialize the object's attributes accordingly.

💡 *Can we create more than one object to a class?*

Yes, the main concept of OOPs is class acts as a template for the objects. So, any number of objects can be created for one class.

Ex 13.9 (Parameterized Constructor)

```
class Television:
   def __init__(self, brand, model, screen_size):
       self.brand = brand
       self.model = model
       self.screen_size = screen_size
       print("This TV is", brand, screen_size,'inch', model)

salon_tv = Television("Sony", "Class X80K", 65)
bedroom_tv = Television("Samsung", "Class N5300 Series", 32)
```

```
This TV is Sony 65 inch Class X80K
This TV is Samsung 32 inch Class N5300 Series
```

What is the difference between a method and constructor?

Method	Constructor
Methods are used to do operations or actions.	Constructors are used to initialize the instance variables.
Method name can be any name.	Constructor name should be --init--
Methods we should call explicitly to execute.	Constructor automatically executed at the time of object creati

`EXERCISE 13.2`

`Create a class called "BankAccount" with customer_name and account_number properties and calculate_balance() method?`

```
# write your answer code here

```

Lesson Three : Class Variables and Methods

What is Class Variable?

Class variables are variables that are defined within a class.

Types of Class Variables in Python:

Inside a class, we can have three types of variables. They are:

- Instance variables (object level variables)
- Static variables (class level variables)
- Local variables

Instance Variables

If the value of a variable is changing from object to object then such variables are called as instance variables.

💡 *Where instance variables can be declared?*

We can declare and initialize the instance variables in following ways,

1. By using a constructor.
2. By using instance methods.
3. By using object name

💡 *Since different TV have different brand, model & screen size, so we need to define variable during object creation*

1. Declaring instance variables inside a constructor

We can declare and initialize instance variables inside a constructor by using self variable.

Every object in Python has an attribute denoted as `--dict--`, which python creates internally. This displays the object information in dictionary form. It maps the attribute name to its value.

Ex 13.10 (Declaring instance variables in a constructor)

```
class Television:
   def __init__(self, brand, model, screen_size):
       self.brand = brand
       self.model = model
       self.screen_size = screen_size

salon_tv = Television("Sony", "Class X80K", 65)
bedroom_tv = Television("Samsung", "Class N5300 Series", 32)
print(salon_tv.__dict__)
print(bedroom_tv.__dict__)
```

```
{'brand': 'Sony', 'model': 'Class X80K', 'screen_size': 65}
{'brand': 'Samsung', 'model': 'Class N5300 Series', 'screen_size': 32}
```

What is Method?

Method is a function that is associated with a specific object or class.

Types of Methods in a Class

In python, we can classify the methods into three types from the perspective of object oriented programming.

1. Instance Methods
2. Class Methods
3. Static Methods

1. Instance Methods

Instance methods are methods which act upon the instance variables of the class. They are bound with instances or objects, that's why called as instance methods. The first parameter for instance methods should be self variable which refers to instance. Along with the self variable it can contain other variables as well.

What is the difference between method and function?

Functions which are written inside a class can be accessed only with the objects created from the class. They are called methods. We can say that methods are nothing but functions which can be referenced by the objects of the class.

💡 *Suppose display and resolution attributes are not part of object creation but display info method, so you need to declare these variables inside a method.*

2. Declaring instance variables inside instance methods

We can declare and initialize instance variables inside instance method by using self variable.

Ex 13.11 (Declaring instance variables inside instance methods)

```
class Television:
  def __init__(self, brand, model, screen_size):
      self.brand = brand
      self.model = model
      self.screen_size = screen_size

  def display_details(self):
      self.resolution = "4K"
      self.display = "LED"

      print("TV Details")
      print("Model :", self.brand, self.model)
      print("Screen Size :", self.screen_size, "inch")
      print("Display :", self.resolution, self.display)
```

```
my_tv = Television("Sony", "Class X80K", 65)
my_tv.display_details()
print(my_tv.__dict__)
```

```
TV Details
Model : Sony Class X80K
Screen Size : 65 inch
Display : 4K LED
{'brand': 'Sony', 'model': 'Class X80K', 'screen_size': 65, 'resolution': '4K', 'displa
```

💡 *Price is not part of the TV device and the same brand & model TV can be sold with different price in different places, so we need to declare price using object name.*

3. Declaring instance variables inside object name

We can declare and initialize instance variables by using object name as well

Ex 13.12 (Declaring instance variables inside object name)

```
class Television:
  def __init__(self, brand, model, screen_size):
      self.brand = brand
      self.model = model
      self.screen_size = screen_size

  def display_details(self):
      self.resolution = "4K"
      self.display = "LED"

      print("TV Details")
      print("Model :", self.brand, self.model)
      print("Screen Size :", self.screen_size, "inch")
      print("Display :", self.resolution, self.display)

my_tv = Television("Sony", "Class X80K", 65)
my_tv.price = 2400
print(my_tv.__dict__)
```

```
{'brand': 'Sony', 'model': 'Class X80K', 'screen_size': 65, 'price': 2400}
```

Accessing instance variables in Python:

The instance variable can be accessed in two ways:

- By using self variable
- By using object name

Ex 13.13 (Accessing instance variables)

```
class Television:
  def __init__(self, model):
      self.brand = "Sony"
      self.model = model

  def display_details(self):
      print("TV Model :", self.model) # accessing instance variable using self

my_tv = Television("Class X80K")
my_tv.display_details()
print("This TV brand is", my_tv.brand) # accessing instance variable using object name
```

```
TV Model : Class X80K
This TV brand is Sony
```

💡 *What if you want to use a variable across all methods of a class?*

Static Variables

If the value of a variable is not changing from object to object, such types of variables are called static variables or class level variables. We can access static variables either by class name or by object name. Accessing static variables with class names is highly recommended than object names.

Ex 13.14 (Accessing static variables outside of class)

```
class Television:
  display = "LED"

print("The TV display is", Television.display) # accessing static variable using class name

my_tv = Television()
print("The TV display is", my_tv.display)      # accessing static variable using object name
```

```
The TV display is LED
The TV display is LED
```

Declaring Static Variables

We can declare static variable in the following ways,

- Inside class and outside of the method
- Inside constructor
- Inside instance method
- Inside class method
- Inside static method

1. Declaring static variable inside class and outside of the method

Generally, we can declare and initialize static variable within the class and outside of the methods. This is the preferred way.

Ex 13.15 (Declaring static variable inside class and outside of the method)

```
class Television:
  display = "LED"

print(Television.__dict__)
```

```
{'__module__': '__main__', 'display': 'LED', '__dict__': <attribute '__dict__' of 'Telev
```

2. Declaring static variable inside constructor

We can declare and initialize static variables within the constructor by using class's name.

Ex 13.16 (Declaring static variable inside constructor)

```
class Television:
  display = "LED"

  def __init__(self):
    Television.brand = "Sony"

my_tv = Television()
print(Television.__dict__)
```

```
{'__module__': '__main__', 'display': 'LED', '__init__': <function Television.__init__ a
```

3. Declaring static variable inside instance method

We can declare and initialize static variable inside instance method by using class name just as we have done in the constructor.

Ex 13.17 (Declaring static variable inside instance method)

```
class Television:
  display = "LED"

  def __init__(self):
    Television.brand = "Sony"

  def mute(self):
    Television.volume = 0
    print("Volume :", Television.volume)
```

```
my_tv = Television()
my_tv.mute()
print(Television.__dict__)
```

```
Volume : 0
{'__module__': '__main__', 'display': 'LED', '__init__': <function Television.__init__ a
```

4. Declaring static variable inside class method

We can declare and initialise static variable inside class method in two ways,

1. using class name,
2. using `cls` pre-defined variable

What is class method?

Class methods are methods which act upon the class variables or static variables of the class. We can go for class methods when we are using only class variables (static variables) within the method.

Class methods should be declared with `@classmethod`.

Just as instance methods have `self` as the default first variable, class method should have `cls` as the first variable. Along with the cls variable it can contain other variables as well.

Ps. Class methods are rarely used in python.

Ex 13.18 (Declaring static variable inside class method)

```
class Television:
  display = "LED"

  def __init__(self):
    Television.brand = "Sony"

  def mute(self):
    Television.volume = 0
    print("Volume :", Television.volume)

  @classmethod
  def change_mode(cls):
    Television.mode = "AV"
    print("Mode :", Television.mode)

my_tv = Television()
my_tv.change_mode()
print(Television.__dict__)
```

```
Mode : AV
{'__module__': '__main__', 'display': 'LED', '__init__': <function Television.__init__ a
```

We can also do the initialization in the class method using the `cls` variable. cls is predefined variable in python. We should pass the cls parameter to the class methods just as we use self for instance methods.

Ex 13.19 (static variable inside class method by using `cls`)

```
class Television:
  display = "LED"

  def __init__(self):
    Television.brand = "Sony"

  def mute(self):
    Television.volume = 0
    print("Volume :", Television.volume)

  @classmethod
  def change_mode(cls):
    Television.mode = "AV"
    print("Mode :", Television.mode)

  @classmethod
  def display_details(cls):
    cls.model = "Class X80K"
    print("TV Model :", Television.model)

my_tv = Television()
my_tv.display_details()
print(Television.__dict__)
```

```
TV Model : Class X80K
{'__module__': '__main__', 'display': 'LED', '__init__': <function Television.__init__ a
```

5. Declaring static variable inside static method

We can declare and initialize static variables inside a static method by using class name.

3. Static Method

Static methods are extremely similar to python class level methods, the difference being that a static method is bound to a class rather than the objects for that class. This means that a static method can be called without an object for that class. This also means that static methods cannot modify the state of an object as they are not bound to it.

Advantages of Python static method

- Static methods have a very clear use-case. When we need some functionality not w.r.t an Object but w.r.t the complete class, we make a method static. This is pretty much advantageous when we need to create Utility methods as they aren't tied to an object lifecycle usually. Finally, note that in a static method, we don't need the self to be passed as the first argument.
- No arguments like `cls` or self are required at the time of declaration.
- We can declare static method explicitly by using `@staticmethod` decorator.
- We can access static methods by using class name or object reference.

Before method if we write `@staticmethod` decorator then that method will become a staticmethod.

Ex 13.20 (static variable inside static method by using class name)

```
class Television:
  display = "LED"

  def __init__(self):
    Television.brand = "Sony"

  def mute(self):
    Television.volume = 0
    print("Volume :", Television.volume)

  @classmethod
  def change_mode(cls):
    Television.mode = "AV"
    print("Mode :", Television.mode)

  @classmethod
  def display_details(cls):
    cls.model = "Class X80K"
    print("TV Model :", Television.model)

  @staticmethod
  def connect_to_netflix():
    Television.web_url = "https://netflix.com"
    print("Connecting to Netflix...")

Television.connect_to_netflix()
print(Television.__dict__)
```

```
Connecting to Netflix...
{'__module__': '__main__', 'display': 'LED', '__init__': <function Television.__init__ a
```

Accessing a static variable

We have already discussed with examples, how static variables can be accessed outside of class earlier. Now, we shall discuss how we can access static variable in (within the class),

- Inside constructor
- Inside instance method
- Inside class method
- Inside static method

1. Accessing a static variable Inside Constructor:

We can access static variables inside the constructor by using either self or class name.

Ex 13.21 (Accessing static variable Inside Constructor)

```
class Television:
  brand = "Sony"
  model = "Class X80K"
  display = "LED"
  resolution = "4K"
  width = 16
  height = 9
  netflix_web_url = "https://netflix.com"

  def __init__(self):
    # access inside a constructor
    print("Display :", self.display, Television.resolution)

  def aspect_ratio(self):
    #access inside instance method
    print("Aspect Ratio is", self.width,":", Television.height)

  @classmethod
  def display_details(cls):
    # access inside class method
    print("TV Model :", cls.brand, Television.model)

  @staticmethod
  def connect_to_netflix():
    # access inside static method
    print("Connecting to", Television.netflix_web_url)

my_tv = Television()
my_tv.aspect_ratio()
my_tv.display_details()
my_tv.connect_to_netflix()
```

```
Display : LED 4K
Aspect Ratio is 16 : 9
TV Model : Sony Class X80K
Connecting to https://netflix.com
```

Local Variables

The variable which we declare inside of the method is called a local variable. Generally, for temporary usage we create local variables to use within the methods. The scope of these variables is limited to the method in which they are declared. They are not accessible out side of the methods.

Ex 13.22 (Local Variables in Python)

```
class Television:
  volume = 15

  def volume_up(self):
    min_volume, max_volume = 0, 100 #local variable

    if Television.volume == max_volume:
      Television.volume == max_volume
    else:
      Television.volume += 1
    print("Volume :", Television.volume)

my_tv = Television()
my_tv.volume_up()
```

```
Volume : 16
```

Ex 13.23 (Accessing Local Variables Outside a Function NameError)

```
class Television:
  volume = 15

  def volume_up(self):
    min_volume, max_volume = 0, 100 #local variable

    if Television.volume == max_volume:
      Television.volume == max_volume
    else:
      Television.volume += 1
    print("Volume :", Television.volume)

  def volume_down(self):
    if Television.volume == min_volume: # 'min_volume' is local variable of volume_up()
      Television.volume == min_volume
    else:
      Television.volume += 1
    print("Volume :", Television.volume)

my_tv = Television()
my_tv.volume_down()
```

```
------------------------------------------------------
--------------------
NameError                                 Traceback
(most recent call last)
<ipython-input-17-1946f5a9f7e8> in <cell line: 21>()
     19
     20 my_tv = Television()
---> 21 my_tv.volume_down()

<ipython-input-17-1946f5a9f7e8> in volume_down(self)
     12
     13   def volume_down(self):
---> 14     if Television.volume == min_volume:
     15       Television.volume == min_volume
     16     else:

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

EXERCISE 13.3

Create a class called "BankAccount" with properties "bank_name" as astatic variable, "balance" as an instance variable and calculate_balance() method using 7% interest rate as a local variable?

```
1 # write your answer code here
2
```

💡 *What if you want to check the existence of a channel before changing it?*

Setter and Getter methods in Python:

Setter methods can be used to set values to the instance variables. They are also known as mutator methods. Their main aim is to only set values to instance variables, hence they don't return anything.

> Syntax:
> `def set_variable(self, variable_name):`
> `    self.variable_name = variable_name`

Getter methods are used to get the values of instance variables. They return the value in the instance variable. Getter methods also known as accessor methods.

> Syntax:
> `def get_variable(self):`
> `    return self.variable_name`

Ex 13.24 (Setter and Getter Methods)

```
class Television:
  def __init__(self):
    self.channels = {1:"Aljezzira", 2:"BBC", 3:"CNN"}

  def change_channel(self, channel):
    if channel >= 0 and channel < len(self.channels):
        self.channel = channel
        channel_name = self.get_channel()
        print("Now you are watching", channel_name)
    else:
        print("Channel does not exist!")

  def get_channel(self):
        return self.channels[self.channel]

my_tv = Television()
my_tv.change_channel(2)

my_tv = Television()
my_tv.change_channel(5)
```

```
Now you are watching BBC
Channel does not exist!
```

EXERCISE 13.4

Create an account number validation class that enforces the following rules:

- The account number must be 6 digits long.
- The account number must start with the digits '10'.

Example : 100345

Implement this using setter and getter methods.

```
# write your answer code here

```

Lesson Four : Encapsulation

What is Encapsulation?

Encapsulation is a way to restrict the direct access to some components of an object, so users cannot access state values for all of the variables of a particular object.

Encapsulation can be used to hide both variables and methods associated with an instantiated class or object.

Benefits of Encapsulation

Encapsulation in programming has a few key benefits. These include:

- **Data Hiding:** Encapsulation allows us to hide internal implementation details from the user, which makes our code more robust and less prone to errors.
- **Code Reusability:** Encapsulated code can be easily reused, tested, and debugged.
- **Security:** Encapsulation provides an extra layer of security by preventing unauthorized access to sensitive data.

Encapsulation is achieved by using access modifiers. However, Python does not have explicit access modifiers like public, private, and protected that are found in other object-oriented programming languages like Java or C++.

- **Public:** Attributes and methods that are accessible from anywhere.
- **Protected:** Attributes and methods that are accessible within the class and its subclasses. They are denoted by a single underscore prefix.
- **Private:** Attributes and methods that are accessible only within the class. They are denoted by a double underscore prefix.

In Python, when you want to make a variable or method "protected" or "private" within a class, you can give it a special name. This special name is called "name mangling".

What is Name Mangling?

Name mangling is a technique used in Python to create a form of name convention for variables and methods that are intended to be "protected" or "private" within a class.

In Python, the convention is to use a single leading underscore (_) to indicate that a variable or method is "**protected**", and a double leading underscore (__) to indicate that it's "**private**".

PS. You can modify both protected and private variables outside the class but can't access only private variable, Which is weird. 🤔

Ex 13.25 (Creating public, protected and private attributes)

```
class Television:
    def __init__(self):
        self.channel = "BBC"  # Public attribute
        self._volume = 15     # Protected attribute
        self.__brand = "Sony"      # Private attribute

    def turn_on(self):
        print(f"The {self.__brand} television is now turned on.")

    def _change_channel(self, channel):
        print(f"The channel has been changed to {channel}.")

    def __adjust_volume(self, volume):
        print(f"The volume has been adjusted to {volume}.")

my_tv = Television()
print(my_tv)
```

```
<__main__.Television object at 0x7e35f6692b90>
```

Ex 13.26 (Calling protected method outside a class)

```
my_tv._change_channel("CNN")
```

```
The channel has been changed to CNN.
```

Ex 13.27 (Calling private method outside a class)

```
my_tv.__adjust_volume(30)
```

```
---------------------------------------------------------
--------------------
AttributeError                            Traceback
(most recent call last)
<ipython-input-5-088d40274ff1> in <cell line: 2>()
      1 # calling private method outside a class
----> 2 my_tv.__adjust_volume(30)

AttributeError: 'Television' object has no attribute
```

Ex 13.28 (Modifying protected and private variables outside a class)

```
my_tv._volume = 0
print(my_tv._volume)

my_tv.__brand = "Samsung"
print(my_tv.__brand)
```

```
0
Samsung
```

The important note here is that name mangling is all about safety and not security. It will not keep you protected against intentional wrongdoing; just, accidental overriding.

Ex 13.29 (Accessing protected variable outside a class)

```
print(my_tv._volume)
```

```
0
```

Ex 13.30 (Accessing private variable outside a class)

```
print(my_tv.__brand)
```

```
Samsung
```

As you can see, attempting to access the private variable from outside the class results in an AttributeError.

One way to access it, if that's necessary, is to create a getter method inside a class that would return the value of the private variable.

Ex 13.31 (Accessing private variable using getter method)

```
class Television:
    def __init__(self):
        self.channel = "BBC"   # Public attribute
        self._volume = 15      # Protected attribute
        self.__brand = "Sony"  # Private attribute

    def turn_on(self):
        print(f"The {self.__brand} television is now turned on.")

    def _change_channel(self, channel):
        print(f"The channel has been changed to {channel}.")

    def __adjust_volume(self, volume):
        print(f"The volume has been adjusted to {volume}.")

    def get_brand(self):
        return self.__brand

my_tv = Television()
print(my_tv.get_brand())
```

```
Sony
```

EXERCISE 13.5

Create a "BankAccount" class that has a "balance" property, and ensure that the balance property cannot be modified outside the class?

```
# write your answer code here

```

Nested Classes in Python:

A class inside another class are called nested classes.

> Syntax:
> ```
> class OuterClass:
>     def __ init__(self):
>         self.NestedClass = self.NestedClass(self)
>     // Outer class variables & method
>
>     class NestedClass:
>         // Nested class constructor, variables & method
> ```

To create an inner class object, first create an outer class object, then use it to create the inner class object and access its members.

Ex 13.32 (Nested Classes)

```
class Television:
    def __init__(self):
        self.volume = 15
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}
        self.remote = self.Remote(self)

    def turn_on(self):
        print("The TV is on.")

    class Remote:
        def __init__(self, television):
            self.battery_percentage = 100
            self.television = television

        def change_channel(self, channel):
            print(f"Changing channel to {self.television.channels[channel]}")

        def increase_volume(self):
            self.television.volume = self.television.volume + 1
            print("Increasing volume:", self.television.volume)

```

```
tv = Television()
tv.turn_on()

# Using the nested Remote class
tv.remote.change_channel(2)
tv.remote.increase_volume()
```

```
The TV is on.
Changing channel to BBC
Increasing volume: 16
```

EXERCISE 13.6

Create a nested class "SavingAccount" inside "BankAccount" class?

```
# write your answer code here

```

What is Garbage Collection in Python?

In old languages like C++, programmers are responsible for both creation and destruction of objects.

Usually a programmer should take very much care to create objects, and sometimes he may neglect destruction of useless objects.

Due to this negligence, at a certain point of time there may be a chance, total memory can be filled with useless objects which creates memory problems and total application will be down with Out of memory error.

But in Python, internally a program will run in background always to destroy useless objects.

So, the chance of failing a Python program with memory problems are very little.

This program is nothing but Garbage Collector.

Garbage collection is an automatic process that manages the memory occupied by objects.

Objective:

The main objective of Garbage Collector is to destroy useless objects and freeing up memory for future use. If an object does not have any reference variable, then that object is eligible for Garbage Collection.

By default, the Garbage collector is enabled, but we can disable it based on our requirement. In order to do so, we need to import the `gc` module.

Methods:

- `gc.isenabled()` – This method returns True if garbage collector is enabled.
- `gc.disable()` – This function is used to is disable the garbage collector explicitly.
- `gc.enable()` – This function is used to is enable the garbage collector explicitly.

Ex 13.33 (Garbage Collection)

```
import gc
print(gc.isenabled())
gc.disable()
print(gc.isenabled())
gc.enable()
print(gc.isenabled())
```

```
False
False
True
```

Lesson Five : Inheritance

What is Inheritance?

Inheritance, in general, is the passing of properties to someone. In programming languages, the concept of inheritance comes with classes.

Creating new classes from already existing classes is called inheritance.

The existing class is called a super class or base class or parent class.

The new class is called a subclass or derived class or child class.

Inheritance allows sub classes to inherit the variables, methods and constructors of their super class.

Advantages of Inheritance:

- The main advantage of inheritance is code re-usability.
- Time taken for application development will be less.
- Redundancy (repetition) of the code can be reduced.

Implementation of Inheritance in Python:

While declaring subclass, we need to pass super class name into subclass's parenthesis

Ex 13.34 (Implementing Inheritance)

```
class Television:
    def __init__(self):
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def turn_on(self):
        print("The TV is on.")

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")
```

```
class Flat_TV():
    def __init__(self):
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def turn_on(self):
        print("The TV is on.")

    def change_channel(self, channel):
        print(f"Changing channel to {self.channels[channel]}")

    def HDMI_mode(self):
      self.mode = "HDMI"
      print("Mode :", self.mode)

my_tv = Television()
my_tv.turn_on()
my_tv.change_channel(3)

my_flat_tv = Flat_TV()
my_flat_tv.turn_on()
my_flat_tv.change_channel(3)
my_flat_tv.HDMI_mode()
```

```
The TV is on.
Changing channel to CNN
The TV is on.
Changing channel to CNN
Mode : HDMI
```

Types of Inheritance in Python:

There are there types of inheritance, they are:

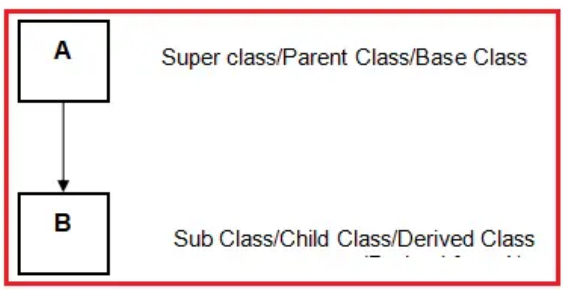
- Single inheritance
- Multilevel inheritance
- Multiple inheritance

💡 *Why would we write same functions in two classes, Since we can inherit it?*

Single Inheritance

Creating a subclass or child class from a single superclass/parent class is called single inheritance.

Diagrammatic representation of Python Single Inheritance is given below.



> Syntax:
>
> ```
> class Parent_Class:
>     // Parent class constructor, variables & method
>
> class Child_Class(Parent_Class):
>     // Child class constructor, variables & method
> ```

Ex 13.35 (Single Inheritance)

```
class Television:
    def __init__(self):
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def turn_on(self):
        print("The TV is on.")

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")

class Flat_TV(Television):
    def HDMI_mode(self):
      self.mode = "HDMI"
      print("Mode :", self.mode)

my_flat_tv = Flat_TV()
my_flat_tv.turn_on()
my_flat_tv.change_channel(3)
my_flat_tv.HDMI_mode()
```

```
The TV is on.
Changing channel to CNN
Mode : HDMI
```

EXERCISE 13.7

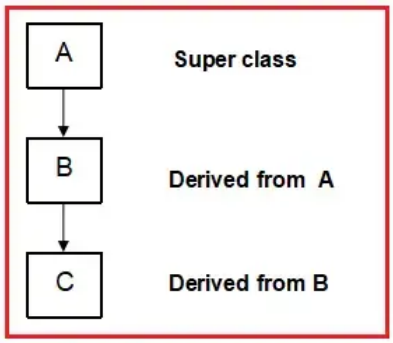
Create "BankHQ" as child class and "BankBranch" as parent class using single inheritance?

```
# write your answer code here

```

Multi-Level Inheritance

If a class is derived from another derived class then it is called multi level inheritance. Diagrammatic representation of Python Multilevel Inheritance is given below.



> Syntax:
>
> ```
> class Grand_Parent_Class:
>     // Grand Parent class constructor, variables & method
>
> class Parent_Class(Grand_Parent_Class):
>     // Parent class constructor, variables & method
>
> class Child_Class(Parent_Class):
>     // Child class constructor, variables & method
> ```

Ex 13.36 (Multilevel Inheritance)

```
class Television:
    def __init__(self):
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def turn_on(self):
        print("The TV is on.")

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")

class Flat_TV(Television):
    def HDMI_mode(self):
      self.mode = "HDMI"
      print("Mode :", self.mode)

class Smart_TV(Flat_TV):
    def connect_to_netflix(self):
      self.web_url = "https://netflix.com"
      print("Connecting to Netflix...")
```

```
my_smart_tv = Smart_TV()
my_smart_tv.turn_on()
my_smart_tv.change_channel(3)
my_smart_tv.HDMI_mode()
my_smart_tv.connect_to_netflix()
```

```
The TV is on.
Changing channel to CNN
Mode : HDMI
Connecting to Netflix...
```

EXERCISE 13.8

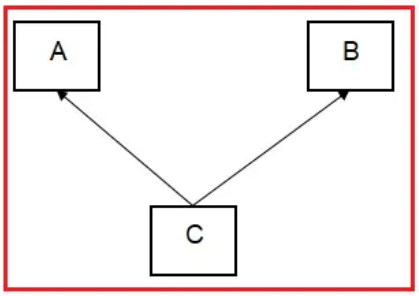
Create "BankHQ" as child class, "BankBranch" as parent class and "BankAccount" as grand parent class using multilevel inheritance?

```
# write your answer code here

```

Multiple Inheritance

If a child class is derived from multiple superclasses then it is called multiple inheritance. Diagrammatic representation of Multiple Inheritance is given below.



> Syntax:
>
> ```
> class Mother_Class:
>     // Mother Parent class constructor, variables & method
>
> class Father_Class:
>     // Father class constructor, variables & method
>
> class Child_Class(Father_Class, Mother_Class):
>     // Child class constructor, variables & method
> ```

💡 *What if you want to connect your smart TV with a smart speaker to enjoy the music clip audio?*

Ex 13.37 (Multiple Inheritance)

```
class Television:
    def __init__(self):
        self.volume = 15
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def turn_on(self):
        print("The TV is on.")

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")

class Smart_Speaker:
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Volume :", self.volume)

class Smart_TV(Television, Smart_Speaker):
    def connect_to_netflix(self):
      self.web_url = "https://netflix.com"
      print("Connecting to Netflix...")

my_smart_tv = Smart_TV()
my_smart_tv.turn_on()
my_smart_tv.change_channel(3)
my_smart_tv.increase_volume()
my_smart_tv.connect_to_netflix()
```

```
The TV is on.
Changing channel to CNN
Volume : 16
Connecting to Netflix...
```

💡 *What if both parent classes have a method with the same name? Suppose the television has a speaker, and the smart speaker also has a speaker. Which speaker's volume are you increasing?*

There may be a chance that two parent classes can contain methods which are having the same method name in both classes. Syntactically this is valid.

Ex 13.38 (Parent classes having same method names)

```
class Television:
    def __init__(self):
        self.volume = 15
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def turn_on(self):
        print("The TV is on.")

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")

    def increase_volume(self):
      self.volume = self.volume + 1
      print("TV Volume :", self.volume)

class Smart_Speaker:
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Speaker Volume :", self.volume)

class Smart_TV(Television, Smart_Speaker):
    def connect_to_netflix(self):
      self.web_url = "https://netflix.com"
      print("Connecting to Netflix...")

my_smart_tv = Smart_TV()
my_smart_tv.turn_on()
my_smart_tv.change_channel(3)
my_smart_tv.increase_volume()
my_smart_tv.connect_to_netflix()
```

```
The TV is on.
Changing channel to CNN
TV Volume : 16
Connecting to Netflix...
```

In the above scenario, you increase the TV volume, not the speaker, because it depends on the order of inheritance of parent classes in the child class.

- `class Smart_TV(Television, Smart_Speaker):` => Television's class method will be considered.
- `class Smart_TV(Smart_Speaker, Television):` => Smart Speaker's class method will be considered.

Ex 13.39 (Order of inheritance)

```
class Television:
    def __init__(self):
        self.volume = 15
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def turn_on(self):
        print("The TV is on.")

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")

    def increase_volume(self):
      self.volume = self.volume + 1
      print("TV Volume :", self.volume)

class Smart_Speaker:
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Speaker Volume :", self.volume)

class Smart_TV(Smart_Speaker, Television):
    def connect_to_netflix(self):
      self.web_url = "https://netflix.com"
      print("Connecting to Netflix...")

my_smart_tv = Smart_TV()
my_smart_tv.turn_on()
my_smart_tv.change_channel(3)
my_smart_tv.increase_volume()
my_smart_tv.connect_to_netflix()
```

```
The TV is on.
Changing channel to CNN
Speaker Volume : 16
Connecting to Netflix...
```

EXERCISE 13.9

Create "SavingAccount" and "CheckingAccount" as child classes of "BankAccount" parent class using multiple inheritance?

```
# write your answer code here

```

💡 *What if both parent and child class have a method with the same name?*

Lesson Six : Polymorphism

What is Polymorphism?

The word 'Poly' means many and 'Morphs' means forms.

Polymorphism is the ability of an object to take on many forms. In other words, polymorphism allows objects of different classes to be treated as if they are of the same class.

Types of Polymorphism in Python:

The following are the examples, or implementations, of polymorphism:

1. Overriding
    - Method Overriding
    - Constructor Overriding
2. Overloading
    - Method Overloading
    - Constructor Overloading
    - Operator Overloading
3. Duck Typing Philosophy

1. Overriding

Overriding refers to the process of implementing something again, which already exists in parent class, in child class.

All the members available in the parent class, those are by-default available to the child class through inheritance. If the child class is not satisfied with parent class implementation, then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding. Overriding concept applicable for both methods and constructors.

1.1. Method Overriding

Method Overriding is the ability of a child class to provide its own implementation of a method that is already defined in its parent class.

In other words, method overriding allows a child class to "override" or replace the implementation of a method that it inherits from its parent class.

Ex 13.40 (Method Overriding)

```
class Smart_Speaker:
    def __init__(self):
        self.volume = 15

    def increase_volume(self):
        self.volume = self.volume + 1
        print(f"Smart Speaker Volume : {self.volume}")

class Smart_TV(Smart_Speaker):
    def increase_volume(self):
        self.volume = self.volume + 1
        print(f"Smart TV Volume : {self.volume}")

my_smart_tv = Smart_TV()
my_smart_tv.increase_volume()
```

```
Smart TV Volume : 16
```

💡 *What if we want to override parent class constructor?*

1.2. Constructor Overriding

If child class does not have constructor, then parent class constructor will be executed at the time of child class object creation. If child class has a constructor, then child class constructor will be executed at the time of child class object creation.

Ex 13.41 (Constructor Overriding)

```
class Smart_Speaker:
    def __init__(self):
        self.volume = 15

    def increase_volume(self):
        self.volume = self.volume + 1
        print(f"Smart Speaker Volume : {self.volume}")

class Smart_TV(Smart_Speaker):
    def __init__(self):
        self.volume = 20

    def increase_volume(self):
        self.volume = self.volume + 1
        print(f"Smart TV Volume : {self.volume}")

my_smart_tv = Smart_TV()
my_smart_tv.increase_volume()
```

```
Smart TV Volume : 21
```

💡 *What if we want to access parent class constructor in child class constructor?*

Super() Function in Python

`super()` is a predefined function in python. By using super() function in child class, we can call,

- Super class constructor.
- Super class methods.
- Super class variables.

Ex 13.42 (Calling super class constructor from child class constructor using `super()`)

```
class Smart_Speaker:
    def __init__(self):
        self.volume = 15
        print("Default Smart Speaker Volume :", self.volume)

class Smart_TV(Smart_Speaker):
    def __init__(self):
        self.volume = 20
        super().__init__()
        print("Smart TV Volume :", self.volume)

my_smart_tv = Smart_TV()
```

```
Default Smart Speaker Volume : 15
Smart TV Volume : 15
```

💡 *What if we want to access parent class method in child class method?*

When both superclass and child class may have the same method names, same variable names, some scenarios may come where we want to use both of them. In such a case, using the super() function we can call the parent class method.

Ex 13.43 (Calling super class method from child class method using `super()`)

```
class Smart_Speaker:
    volume = 15
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Smart Speaker Volume :", self.volume)

class Smart_TV(Smart_Speaker):
    volume = 20
    def increase_volume(self):
      super().increase_volume()
      print("Smart TV Volume :", self.volume)

my_smart_tv = Smart_TV()
my_smart_tv.increase_volume()
```

```
Television Volume : 21
Smart TV Volume : 21
```

💡 *What if we want to access parent class method of nested inheritance in child class method?*

In the case where the grandparent, parent, and child classes all have a method with the same name, **`super(class_name, self).method_name`** will call the method of super class of mentioned class name.

Ex 13.44 (Accessing parent class method of nested inheritance using `super()`)

```
class Speaker:
    volume = 10
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Speaker Volume :", self.volume)

class Smart_Speaker(Speaker):
    volume = 15
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Smart Speaker Volume :", self.volume)

class Smart_TV(Smart_Speaker):
    volume = 20
    def increase_volume(self):
      super(Smart_Speaker, self).increase_volume()
      print("Smart TV Volume :", self.volume)

my_smart_tv = Smart_TV()
my_smart_tv.increase_volume()
```

```
Speaker Volume : 21
Smart TV Volume : 21
```

💡 *What if both parent and child class have a variable with the same name?*

Ex 13.45 (Calling super class variable from child class method using `super()`)

```
class Smart_Speaker:
    volume = 15
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Smart Speaker Volume :", self.volume)

class Smart_TV(Smart_Speaker):
    volume = 20
    def increase_volume(self):
      self.volume = super().volume + 1
      print("Smart TV Volume :", self.volume)

my_smart_tv = Smart_TV()
my_smart_tv.increase_volume()
```

```
Smart TV Volume : 16
```

Different cases for using super() function

- You can access parent class static variables inside child class method.
- You can call parent class instance, static & class methods from child class constructor.
- You can call parent class instance, static & class methods from child class method.
- You can call parent class static method & class method from child class (class method).
- But, you **can't** call parent class instance method & constructor from child class (class method).

Ex 13.46 (Calling super class instance method from child class (class method) using `super()`)

```
class Smart_Speaker:
    volume = 15

    def increase_volume(self):
      self.volume = self.volume + 1
      print("Smart Speaker Volume :", self.volume)

class Smart_TV(Smart_Speaker):
    volume = 20

    @classmethod
    def increase_volume(cls):
      super().increase_volume()
      print("Smart TV Volume :", Smart_TV.volume)

my_smart_tv = Smart_TV()
my_smart_tv.increase_volume()
```

```
---------------------------------------------------------
--------------------
TypeError                                 Traceback
(most recent call last)
<ipython-input-35-0f3ec70c2cdc> in <cell line: 17>()
     15
     16 my_smart_tv = Smart_TV()
---> 17 my_smart_tv.increase_volume()

<ipython-input-35-0f3ec70c2cdc> in increase_volume(cls)
     11     @classmethod
     12     def increase_volume(cls):
---> 13       super().increase_volume()
     14       print("Smart TV Volume :",
Smart_TV.volume)
     15
```

💡 *So, What is the solution for the above issue?*

As discussed above, you **can't** call parent class instance method & constructor from child class (class method). But there is way to access them it using cls keyword inside **`super()`**.

> Syntax:
>
> `super(child_class_name, cls).method_name(cls)`

Ex 13.47 (Calling super class instance method from child class (class method) using `cls`)

```
class Smart_Speaker:
    volume = 15

    def increase_volume(self):
      self.volume = self.volume + 1
      print("Smart Speaker Volume :", self.volume)

class Smart_TV(Smart_Speaker):
    volume = 20

    @classmethod
    def increase_volume(cls):
      super(Smart_TV, cls).increase_volume(cls)
      print("Smart TV Volume :", Smart_TV.volume)

my_smart_tv = Smart_TV()
my_smart_tv.increase_volume()
```

```
Smart Speaker Volume : 21
Smart TV Volume : 21
```

EXERCISE 13.10

You are building a banking application and need to create a hierarchy of bank account classes.

Requirements:

1. Implement a "BankAccount" class with the following:

- Properties: account_number, balance
- Methods: deposit(amount), withdraw(amount), get_balance()

2. Implement a "SavingAccount" class that inherits from "BankAccount" and has the following additional:

- Property: interest_rate
- Method: calculate_interest() (calculates the interest earned on the account balance)
- Method: withdraw(amount) (overridden to handle withdrawal limit)

```
# write your answer code here

```

💡 *What if we have two methods with same name in a single class?*

2. Overloading

Overloading in Python is an implementation where a class can have multiple methods with the same name, but with different parameters.

There are 3 types of overloading:

- Method Overloading
- Constructor Overloading
- Operator Overloading

2.1. Method Overloading

Method overloading is when a class has multiple methods with the same name, but with different parameters.

But in Python Method overloading is not possible. If we are trying to declare multiple methods with the same name and different number of arguments, then Python will always consider only the last method.

Ex 13.48 (Method Overloading)

```
class Television:
    def __init__(self):
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN", 4: "FIFA TV"}
        self.channel = 3

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing TV channel to {self.channels[self.channel]}")

    def change_channel(self, command):
        self.command = command
        if self.command == "Up":
            self.channel += 1
            print(f"Changing Smart TV channel to {self.channels[self.channel]}")
        elif self.command == "Down":
            self.channel -= 1
            print(f"Changing Smart TV channel to {self.channels[self.channel]}")
        else:
            print("You have pressed wrong button!")

my_tv = Television()
my_tv.change_channel(1)
my_tv.change_channel("Up")
```

```
You have pressed wrong button!
Changing Smart TV channel to FIFA TV
```

💡 *How we can handle overloaded method requirements in Python?*

Most of the time, if a method with a variable number of arguments is required then we can handle it with default arguments or with a variable length of argument methods.

Ex 13.49 (Default Arguments in Method Overloading)

```
class Television:
    def __init__(self):
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN", 4: "FIFA TV"}
        self.channel = 3

    def change_channel(self, command):
        if type(command) == int:
          self.channel = command
          print(f"Changing TV channel to {self.channels[self.channel]}")
        elif type(command) == str:
          self.command = command
          if self.command == "Up":
            self.channel += 1
            print(f"Changing TV channel to {self.channels[self.channel]}")
          elif self.command == "Down":
            self.channel -= 1
            print(f"Changing TV channel to {self.channels[self.channel]}")
          else:
            print("You have pressed wrong button!")
        else:
          print("Unavailable command!")

my_tv = Television()
my_tv.change_channel(1)
my_tv.change_channel("Up")
my_tv.change_channel("Down")

my_tv.change_channel(2.0)
my_tv.change_channel("Select")
```

```
Changing TV channel to Aljezzira
Changing TV channel to BBC
Changing TV channel to Aljezzira
Unavailable command!
You have pressed wrong button!
```

2.2. Constructor Overloading

Constructor overloading is also not possible in Python. If we define multiple constructors, only the last constructor will be considered.

Ex 13.50 (Constructor Overloading)

```
class Television:
    def __init__(self, mode):
        self.mode = mode
        print(f"TV Mode : {self.mode}")

    def __init__(self, channel, volume):
        self.channel = channel
        self.volume = volume
        print(f"Now, You are watching {self.channel}")
        print(f"Volume of TV: {self.volume}")

my_tv = Television("BBC", 15)
my_tv = Television("AV")
```

```
Now, You are watching BBC
Volume of TV: 15
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-57303b3aff61> in <cell line: 13>()
     11
     12 my_tv = Television("BBC", 15)
---> 13 my_tv = Television("AV")

TypeError: Television.__init__() missing 1 required positional argument: 'volume'
```

In the above program only constructor with two arguments is working. But based on our requirement we can declare constructor with default arguments and variable length arguments.

Ex 13.51 (Default Arguments in Constructor Overloading)

```
class Television:
    def __init__(self, mode=None, channel=None, volume=None):
        self.mode = mode
        self.channel = channel
        self.volume = volume
        print(f"TV Mode : {self.mode}")
        print(f"Now, You are watching {self.channel}")
        print(f"Volume of TV: {self.volume}")

my_tv = Television(mode="AV")
print("\n")
my_tv = Television(channel="BBC", volume=15)
```

```
TV Mode : AV
Now, You are watching None
Volume of TV: None
```

```
TV Mode : None
Now, You are watching BBC
Volume of TV: 15
```

2.3. Operator Overloading

If we use the same operator for multiple purposes, then it is nothing but operator overloading.

'+' addition operator can be used for Arithmetic addition and String concatenation as well '*' multiplication operator can be used for multiplication for numbers and repetition for strings, lists, tuples etc

Ex 13.52 (' + ' Operator Overloading)

```
1 print(10 + 20)
2 print("Python" + "Programming")
3 print([1, 2, 3] + [4, 5, 6])
```

```
30
PythonProgramming
[1, 2, 3, 4, 5, 6]
```

Ex 13.53 (' * ' Operator Overloading)

```
1 print(10 * 20)
2 print("Python" * 3)
3 print([1, 2, 3] * 3)
```

```
200
PythonPythonPython
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

💡 *What if you want to pass an object as a function parameter?*

3. Duck Typing Philosophy of Python

Duck typing refers to the programming style, in which the object passed to a method supports all the attributes expected from it. The important thing here is not about the object type, but about what attributes and methods the object supports.

In duck typing, while creating a data, it's not required to declare the argument type explicitly. Based on provided value the type will be considered automatically. Since Python is considered as a Dynamically Typed Programming Language, it follows Duck Typing.

PS. "If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

💡 *Imagine you want to increase volume of both TV and Speaker with a single common remote control?*

Ex 13.54 (Duck typing philosophy)

```
class Television:
    volume = 20
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Smart TV Volume :", self.volume)

class Speaker:
    volume = 15
    def increase_volume(self):
      self.volume = self.volume + 1
      print("Smart Speaker Volume :", self.volume)

class Remote:
    def volume_up(self, object):
        object.increase_volume()

my_remote_control = Remote()
my_tv = Television()
my_remote_control.volume_up(my_tv)

my_speaker = Speaker()
my_remote_control.volume_up(my_speaker)
```

```
Smart TV Volume : 21
Smart Speaker Volume : 16
```

In the above program, the function **`volume_up()`** takes an object and calls for the **`increase_volume()`** method of it. With duck typing, the function is not worried about what object type of object it is. The only thing that matters is whether the object has a method with name **`increase_volume()`** supported or not.

EXERCISE 13.11

You are working on a banking application and need to implement a system that can handle different types of accounts (e.g., BankAccount, DebitCard) using a common interface.

Requirements:

1. Implement a "BankAccount" class with the following:

- `Properties: account_number, balance`
- `Methods:`
    - `deposit(amount): Deposits the specified amount into the account.`
    - `withdraw(amount): Withdraws the specified amount from the account if the balance is sufficient.`
    - `get_balance(): Returns the current balance of the account.`

2. Implement a "DebitCard" class with the following:

- Properties: account_number, balance
- Methods:
    - deposit(amount): Deposits the specified amount into the account.
    - withdraw(amount): Withdraws the specified amount from the account if the balance is sufficient.
    - get_balance(): Returns the current balance of the account.

3. Implement a "perform_transaction()" function that takes an account object (either "BankAccount" or "DebitCard") and the following parameters:

- amount: The amount to be deposited or withdrawn.
- transaction_type: The type of transaction ("deposit" or "withdraw").
- The function should perform the specified transaction and print the current balance of the account.

```
# write your answer code here

```

Lesson Seven : Abstraction

What is an Abstraction?

Abstraction is the process of simplifying complex things by focusing on the important parts and hiding unnecessary details. It allows us to create models or representations of real-world objects or concepts that capture their essential characteristics.

Just like we use a remote control to operate a TV without knowing the internal circuitry, abstraction lets us use objects and functions in our code without understanding how they are implemented behind the scenes. We work with a higher-level view that provides the necessary functionality while hiding the complexity.

Abstraction helps programmers to organize code, make it easier to understand, and promote reusability. It allows us to focus on what needs to be done rather than how it's done, enabling efficient problem-solving and creating more maintainable and flexible software.

Types of Methods in Python:

Based on the implementation the methods can be divided into two types:

- Implemented methods
- Un-implemented methods

1. Implemented methods:

A method which has a both method name and method body, that method is called an implemented method. They are also called concrete methods or non-abstract methods. The methods which we have seen in the previous chapters are implemented i.e having method name and body as well.

2. Un-implemented methods:

A method which has only method name and no method body, that method is called an unimplemented method. They are also called as non-concrete or abstract methods.

What is an Abstract Method?

An abstract method is a method that is declared in a class, but has no implementation. It is a method that has no body and exists only as a declaration.

How to declare an abstract method?

Abstract methods, in python, are declared by using `@abstractmethod` decorator. `@abstractmethod` decorator presents in `abc` module.

> Syntax:

```
from abc import ABC, abstractmethod

class Class_Name:
    def --init--(self):
        // body of constructor

    @abstractmethod
    def Unimplemented_Method_Name(self):
        pass

    def Implemented_Method_Name(self):
        print("implemented method")
```

What is an Abstract Class?

An abstract class is like a general template or blueprint for a group of related objects. It defines common characteristics and behaviors that these objects should have. However, an abstract class itself cannot be directly used to create objects.

Class which contains one or more abstract methods is called an abstract class. Abstract class can contain Constructors, Variables, abstract methods, non-abstract methods, and Subclass.

PS. A method which has implementation in the abstract class is called as Concrete Method.

💡 *Let's consider the example of a television. A television can have various types like Smart TV, LED TV, or Plasma TV. While they may have some common features like powering on, powering off, and changing channels, each type of television can have its own specific behaviors or additional features.*

Rules of Abstract Class

- We should import the abc module in order to use the decorator.
- Since abstract method is an unimplemented method, we need to put a pass statement, else it will result in error.
- Abstract methods should be implemented in the subclass or child class of the abstract class.
- If the implementation of the abstract method is not provided in subclass, then that subclass, automatically, will become an abstract class.
- Then, if any class is inheriting this subclass, then that subclass should provide the implementation for abstract methods.
- An abstract class can contain more than one subclass. If different child classes require different kinds of implementations, in that case an abstract class can contain more than one subclass.
- Object creation is not possible for abstract class.
- If any class is inheriting ABC class, and that class doesn’t contain an abstract method, then we can create an object to that class.
- If any class is not inheriting ABC class, then we can create an object for that class even though it contains an abstract method.
- We can create objects for child classes of abstract classes to access implemented methods.

Ex 13.55 (Abstract Class and Abstract Method)

```
from abc import ABC, abstractmethod

class Television(ABC):
    def __init__(self):
        self.volume = 15

    def power_on(self):
        print("The TV is powered on.")

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

```
class SmartTV(Television):
    def increase_volume(self):
        self.volume += 1
        print("Smart TV Volume :", self.volume)

    def decrease_volume(self):
        self.volume -= 1
        print("Smart TV Volume :", self.volume)

my_smart_tv = SmartTV()
my_smart_tv.power_on()
my_smart_tv.increase_volume()
my_smart_tv.decrease_volume()
```

```
The TV is powered on.
Smart TV Volume : 16
Smart TV Volume : 15
```

Ex 13.56 (Abstraction in Multilevel Inheritance)

```
from abc import ABC, abstractmethod

class Television(ABC):
    def __init__(self):
        self.volume = 15
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def power_on(self):
        print("The TV is powered on.")

    @abstractmethod
    def increase_volume(self):
        pass

    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")

class SmartTV(Television):
    def increase_volume(self):
        self.volume += 1
        print("Smart TV Volume :", self.volume)

    def decrease_volume(self):
        self.volume -= 1
        print("Smart TV Volume :", self.volume)

class Remote(SmartTV):
    def change_channel(self, channel):
        self.channel = channel
        print(f"Changing channel to {self.channels[self.channel]}")
```

```
my_tv_remote = Remote()
my_tv_remote.power_on()
my_tv_remote.change_channel(2)
```

```
The TV is powered on.
Changing channel to BBC
```

💡 *When should we go for abstract class?*

An abstract class is a class which can contains few implemented methods and few unimplemented methods as well. When we know about requirements partially, but not completely, then we should go for abstract class.

💡 *When should we go for concrete class?*

Concrete class is a class which is fully implemented. It contains only implemented methods. When we know complete implementation about requirements, then we should go for concrete class.

What is Interface in Python?

We have discussed that an abstract class is a class which may contain some abstract methods as well as non-abstract methods also. Imagine there is an abstract class which contains only abstract methods and doesn't contain any concrete methods, such classes are called Interfaces. Therefore, an interface is nothing but an abstract class which can contains only abstract methods.

Points to Remember:

- In python there is no separate keyword to create an interface.
- We can create interfaces by using abstract classes which have only abstract methods.
- Interface can contain constructors, variables, abstract methods and child class.
- As we know, abstract methods should be implemented in the subclass of interface (Abstract Class).
- Object creation is not possible for the interface class (abstract class).
- We can create objects for the child class of interface to access implemented methods.

Ex 13.57 (Implementation of Interface)

```
from abc import ABC, abstractmethod
class Television(ABC):
    def __init__(self):
        self.volume = 15
        self.channels = {1: "Aljezzira", 2: "BBC", 3: "CNN"}

    def power_on(self):
        pass

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

```
class SmartTV(Television):
    def power_on(self):
        print("The TV is powered on.")

    def increase_volume(self):
        self.volume += 1
        print("Smart TV Volume :", self.volume)

    def decrease_volume(self):
        self.volume -= 1
        print("Smart TV Volume :", self.volume)

my_smart_tv = SmartTV()
my_smart_tv.power_on()
my_smart_tv.increase_volume()
my_smart_tv.decrease_volume()
```

```
The TV is powered on.
Smart TV Volume : 16
Smart TV Volume : 15
```

💡 *When should we go for interfaces?*

Since, Interfaces will not contain implemented methods, when we don't know anything about implementation of requirements, then we should go for interfaces.

`EXERCISE 13.12`

`You are building a banking application and need to create an abstract base class for bank accounts with common functionality.`

`Requirements:`

`1. Implement an abstract "BankAccount" class which has abstract methods: deposit(amount), withdraw(amount), get_balance()`

`2. Implement "SavingsAccount" class inherits from "BankAccount" class which has:`

- `Properties: account_number, balance, interest_rate`
- `Methods:`
  - `deposit(amount): Deposits specified amount into the account and print it.`
  - `withdraw(amount): Withdraws specified amount from the account.`
  - `get_balance(): Returns the current balance of the account.`
  - `calculate_interest(): Calculates the interest earned on the account balance.`

```
# write your answer code here

```

Key Takeaways

- *There are 3 programming paradigms; procedure-oriented, functional-oriented and object-oriented.*
- *Object-oriented programming (OOP) is a way of organizing and designing computer programs.*
- *Class is like a blueprint or a template for creating objects. and object is instance of a class.*
- *Constructor is a special method within a class that is automatically called when an object is created.*
- `self` *is a default variable that refers to a current class object.*
- *Class variables are variables that are defined within a class.*
- *Method is a function that is associated with a specific object or class.*
- *Static variable is a variable that is associated with a class, not with a specific instance of that class.*
- *Class methods are methods which act upon the class variables or static variables of the class.*
- *Static methods are class-level methods that are not bound to specific instances of a class.*
- *Local variables are variable which we declare inside of the method.*
- *Setter and Getter methods are methods which are used to set and get values of instance variables.*
- *Encapsulation is a way to restrict the direct access to some components of an object.*
- *Name mangling is a technique used to create "protected" or "private" attributes within a class.*
- *Nested class is a class inside another class.*
- *Garbage collection is an automatic process that manages the memory occupied by objects.*
- *Inheritance is the process of creating new classes from already existing classes.*
- *There are three types of inheritance; Single, Multiple, Multi-Level.*
- *Polymorphism is the ability of an object to take on many forms.*
- *Overriding is the process of redefining a method in a child class that already exists in the parent class.*
- `super()` *function is a built-in function that allows you to call a method in a parent from a child class.*
- *Overloading is the ability to define multiple methods in a class with the same name but different parameters.*
- *Duck typing is a programming style, in which the object passed to a method supports all the attributes expected from it.*
- *Abstraction is the process of simplifying complex things by focusing on the important parts and hiding unnecessary details.*
- *Interface is an abstract class which contains only abstract methods and doesn't contain any concrete methods.*

// END OF CHAPTER THIRTEEN

Click here for Chapter Fourteen

CHAPTER FOURTEEN

Exception Handling

Objectives

- *What is Error?*
- *What are types of error in programming?*
- *What are Syntax Error?*
- *What are Runtime Error?*
- *What are types of program execution flow?*
- *What is an Exception?*
- *What is the difference between error and exception?*
- *What is Exception Handling?*
- *How to Handle Exception in Python?*
- *What is finally Block?*
- *What are types of exception?*
- *What are Predefined Exceptions?*
- *What are Custom Exceptions?*

Lesson One : Introduction to Exception Handling

Exception handling is a way to deal with unexpected or exceptional situations that can occur while running a program. It helps you handle errors and prevent your program from crashing.

💡 *Think of it this way: Imagine you're following a recipe to bake a cake. As you go step by step, you may encounter unexpected situations like running out of a specific ingredient or accidentally burning the cake. Exception handling is like having a backup plan or alternative actions to take when these unexpected situations happen.*

In programming, similar unexpected situations can occur. For example, you might try to open a file that doesn't exist, divide a number by zero, or receive invalid input from a user. These situations can cause errors, and if not handled properly, they can cause your program to stop working.

What is Error?

An error refers to an unexpected or exceptional condition that occurs during the execution of a program. Errors can arise due to various reasons, such as invalid input, resource unavailability, programming mistakes, or external factors beyond the program's control.

When an error occurs, it disrupts the normal flow of the program and can lead to undesired or incorrect behavior.

To handle these situations, you use exception handling. It allows you to anticipate and catch these errors, and instead of letting your program crash, you can take control and respond in a more controlled and graceful manner.

Types of Error:

In any programming language there are two types of errors. They are:

- Syntax Errors
- Runtime Errors

1. Syntax Errors

The errors which occur because of invalid syntax are called syntax errors.

Ex 14.1 (Syntax Error)

```
1 speed = 113
2 if speed > 113
3    print("Driving beyond the speed limit!")
```

```
      File "<ipython-input-1-46dca1830c3e>", line 2
        if speed > 113
                    ^
    SyntaxError: expected ':'
```

Here we missed to place a colon in the if condition, which is violating the syntax. Hence, syntax error.

The programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

2. Runtime Errors

While executing the program if something goes wrong then we will get Runtime Errors. They might be caused due to,

- End user input
- Programming logic
- Memory problems etc.

Such types of errors are called exceptions.

💡 *Let's say you are working on a program that calculates the average speed of a moving object based on the distance traveled and the time taken.*

The formula for calculating average speed is: `speed = distance / time`

Ex 14.2 (Runtime Error)

```
distance = 120
time = 0
fuel_consumed = 10

speed = distance / time
```

```
---------------------------------------------------------
--------------------
ZeroDivisionError                         Traceback
(most recent call last)
<ipython-input-10-c2a3b7613add> in <cell line: 5>()
      3 fuel_consumed = 10
      4
----> 5 speed = distance / time

ZeroDivisionError: division by zero
```

PS. Exception Handling concept applicable for Runtime Errors but not for syntax errors.

Types of Program Execution Flow

Based on the success of execution, the program execution flow can be categorized into two types:

1. Normal Flow of Program Execution
2. Abnormal Flow of Program Execution

1. Normal Flow of Program Execution

In a program if all statements are executed as per the conditions, successfully and we get output then that flow is called as normal flow of the execution. Below program will get executed successfully from starting to ending.

Ex 14.3 (Normal Flow)

```
distance = 120
time = 2
fuel_consumed = 10

speed = distance / time
print(f"Speed = {speed} mph")
mileage = distance / fuel_consumed
print(f"Mileage = {mileage} MPG")
```

```
Speed = 60.0 mph
Mileage = 12.0 MPG
```

2. Abnormal Flow of Program Execution:

If any error occurs at runtime (executing code) then immediately program flow gets terminated abnormally, without executing the other statements. This kind termination is called an abnormal flow of the execution.

Ex 14.4 (Abnormal Flow)

```
distance = 120
time = 0
fuel_consumed = 10

speed = distance / time
print(f"Speed = {speed} mph")
mileage = distance / fuel_consumed
print(f"Mileage = {mileage} MPG")
```

```
---------------------------------------------------------
--------------------
ZeroDivisionError                         Traceback
(most recent call last)
<ipython-input-13-92efc4d8ad54> in <cell line: 5>()
      3 fuel_consumed = 10
      4
----> 5 speed = distance / time
      6 print(f"Speed = {speed} mph")
      7 mileage = distance / fuel_consumed

ZeroDivisionError: division by zero
```

The above program terminated in the middle where run time error occurred. As discussed, if runtime error occurs it won't execute remaining statements.

What is an Exception?

An unwanted or unexpected event which disturbs the normal flow of the program is called exception. Whenever an exception occurs, then immediately program will terminate abnormally. In order to get our program executed normally, we need to handle those exceptions on high priority.

What is the difference between error and exception?

Errors and exceptions are both problems that can happen in a program. The difference is how they are dealt with.

Errors are major problems that stop the program from running correctly. These are usually caused by bugs in the code or issues with the system the program is running on. Once an error happens, the program cannot continue.

Exceptions are a specific type of problem that can be fixed. They interrupt the normal way the program works. However, the program can catch and handle exceptions. This allows the program to deal with the problem and keep running.

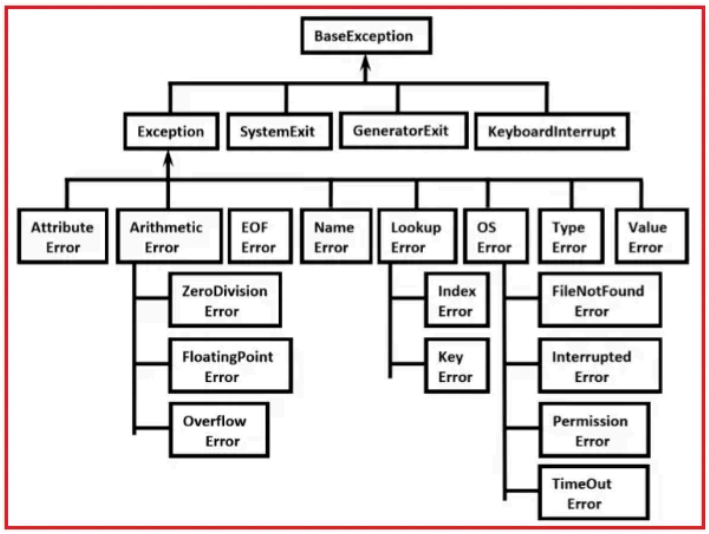
What is Exception Handling?

Exception handling does not mean repairing or correcting the exception. Instead, it is the process in which we define a way, so that the program doesn't terminate abnormally due to the exceptions.

In python, for every exception type, a corresponding class is available and every exception is an object to its corresponding class. Whenever an exception occurs, Python Virtual Machine (PVM) will create the corresponding exception object and will check for handling code.

If handling code is not available, then Python interpreter terminates the program abnormally and prints corresponding exception information to the console. The rest of the program won't be executed.

Exception Hierarchy:



Note: Every Exception in Python is a class. The BaseException class is the root class for all exception classes in python exception hierarchy and all the exception classes are child classes of BaseException. The Programmers need to focus and understand clearly the Exception and child classes.

Lesson Two : Exception Handling Using `try except` Block

How to Handle Exception in Python?

The `try-except` block is used to catch and handle exceptions. It allows you to write code that may potentially raise an exception within the try block and specify how to handle the exception within the except block.

- **`try` block:** try is a keyword in python. The code which may or expected to raise an exception, should be written inside the try block.
- **`except` block:** except is a keyword in python. The corresponding handling code for the exception, if occured, needs to be written inside the except block.

> Syntax:
>
> ```
> try:
>     // code which may raise an exception
> except Exception:
>     // exception handling code
> ```

How it works?

If an exception is raised in the `try` block, the code execution within the `try` block is immediately halted, and the program flow is transferred to the corresponding `except` block that matches the raised exception. Any code that follows the problematic code within the try block will not be executed.

Ex 14.5 (Handling exception by using try and except)

```
distance = 120
time = 0
fuel_consumed = 10

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError:
   print("Exception Passed")

mileage = distance / fuel_consumed
print(f"Mileage = {mileage} MPG")
```

```
Exception Passed
Mileage = 12.0 MPG
```

💡 *What if there is no exception in the `try` block?*

If there is no exception, then should be normal flow with normal termination. In this case, the except block won't execute.

Ex 14.6 (No Exception in the try block)

```
distance = 120
time = 2
fuel_consumed = 10

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError:
   print("Exception Passed")

mileage = distance / fuel_consumed
print(f"Mileage = {mileage} MPG")
```

```
Speed = 60.0 mph
Mileage = 12.0 MPG
```

💡 *What if an exception occurs before* `try` *block?*

If an exception occurs before the try block then the program terminates abnormally. Only the code inside the try block will be checked for the exception and its handling code. But, if there is any exception for the code before the try block, then abnormal termination

Ex 14.7 (Exception occurs before try block)

```
distance = 120
time = 0
fuel_consumed = 0

mileage = distance / fuel_consumed
print(f"Mileage = {mileage} MPG")

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError:
   print("Exception Passed")

print("Fuel Level = Low")
```

```
---------------------------------------------------------
--------------------
ZeroDivisionError                         Traceback
(most recent call last)
<ipython-input-17-482d110ccb2d> in <cell line: 5>()
      3 fuel_consumed = 0
      4
----> 5 mileage = distance / fuel_consumed
      6 print(f"Mileage = {mileage} MPG")
      7

ZeroDivisionError: division by zero
```

EXERCISE 14.1

Write a Python function that takes a list and an index as input, and returns the element at the specified index. If the user tries to access an index that is out of range, the function should catch the IndexError and print an appropriate error message?

```
# write your answer code here

```

Printing Exception Information

In the above examples, in the exception handling code, we are printing our custom message like “Exception Passed” etc. Rather than that, we can also print exception information by creating a reference to the exception.

> Syntax:
>
> `except Exception as e:`

Ex 14.8 (Printing exception information)

```
distance = 120
time = 0
fuel_consumed = 10

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError as e:
  print("Exception Information:", e)

mileage = distance / fuel_consumed
print(f"Mileage = {mileage} MPG")
```

```
Exception Information: division by zero
Mileage = 12.0 MPG
```

EXERCISE 14.2

Write a Python function that takes a dictionary and a key as input, and returns the value associated with the key. If the user tries to access a key that does not exist in the dictionary, the function should catch the KeyError and print an appropriate error message?

```
# write your answer code here

```

💡 *What if two exceptions happened in a single line of code?*

`try` with multiple `except` blocks:

In python, try with multiple except blocks are allowed. Sometimes, there may be the possibility that a piece of code can raise different exceptions in different cases. In such cases, in order to handle all the exceptions, we can use multiple except blocks. So, for every exception type a separate except block we have to write.

> Syntax:
> `try:`
> `    code which may raise an exception`
> `except Exception 1:`
> `    Exception 1 handling code`
> `except Exception 2:`
> `    Exception 1 handling code`

Ex 14.9 (`try` with multiple `except` blocks)

```
def calculate_speed(distance, time):
  try:
    speed = int(distance) / int(time)
    print(f"Speed = {speed} mph")
  except ZeroDivisionError:
    print(f"Can't divide {distance} by {time}.")
  except ValueError:
    print("Please provide int value only!")

calculate_speed(120, 2)
calculate_speed(120, 0)
calculate_speed(120, 'two')
```

```
Speed = 60.0 mph
Can't divide 120 by 0.
Please provide int value only!
```

💡 *Can we handle multiple exceptions in a single except block?*

Single `except` block handling multiple exceptions:

Rather than writing different except blocks for handling different exceptions, we can handle all them in one except block. The only thing is we will not have the flexibility of customizing the message for each exception. Rather, we can take the reference to the exception and print its information as shown in one of the examples above.

> Syntax:
> `except (Exception 1, Exception 2, Exception 3, ...) as e:`

Parenthesis are mandatory, and this group of exceptions is internally considered as tuple.

Ex 14.10 (single `except` block handling multiple exceptions)

```
def calculate_speed(distance, time):
  try:
    speed = int(distance) / int(time)
    print(f"Speed = {speed} mph")
  except (ZeroDivisionError, ValueError) as e:
    print("Please Provide valid numbers only and problem is:", e)

calculate_speed(120, 2)
calculate_speed(120, 0)
calculate_speed(120, 'two')
```

```
Speed = 60.0 mph
Please Provide valid numbers only and problem is: division by zero
Please Provide valid numbers only and problem is: invalid literal for int() with base 10
```

`EXERCISE 14.3`

`Write a Python function that takes a temperature value and a temperature scale (Celsius, Fahrenheit, or Kelvin) as input, and converts the temperature to the other two scales?`

`The function should handle the following exceptions:`

- `ValueError: If the user inputs an invalid temperature scale.`
- `ZeroDivisionError: If the user tries to convert a temperature from Kelvin to Celsius (since the Kelvin scale starts at absolute zero).`

```
# write your answer code here

```

💡 *What if we don't know the exception type occured in the* ***`try`*** *block?*

Default `except` Block:

We can use default except block to handle any type of exceptions. It's not required to mention any exception type for the default block. Whenever we don't have any idea of what expectation the code could raise, then we can go for default except block.

Ex 14.11 (Default `except` block)

```
def calculate_speed(distance, time):
  try:
    speed = int(distance) / int(time)
    print(f"Speed = {speed} mph")
  except ZeroDivisionError:
    print(f"Can't divide {distance} by {time}.")
  except:
    print("Default Exception: Please provide valid input only!")

calculate_speed(120, 2)
calculate_speed(120, 0)
calculate_speed(120, 'two')
```

```
Speed = 60.0 mph
Can't divide 120 by 0.
Default Exception: Please provide valid input only!
```

We know that we can have multiple except blocks for a single try block. If the default except block is one among those multiple ones, then it should be at last, else we get SyntaxError.

Ex 14.12 (SyntaxError in default `except` block)

```
def calculate_speed(distance, time):
  try:
    speed = int(distance) / int(time)
    print(f"Speed = {speed} mph")
  except:
    print("Default Exception: Please provide valid input only!")
  except ZeroDivisionError:
    print(f"Can't divide {distance} by {time}.")

calculate_speed(120, 2)
calculate_speed(120, 0)
calculate_speed(120, 'two')
```

```
  File "<ipython-input-55-211aece4c72e>", line 5
    except:
    ^
SyntaxError: default 'except:' must be last
```

EXERCISE 14.4

Write a Python function that takes a string as input and converts it to an integer. If the user inputs a string that cannot be converted to an integer, the function should catch the ValueError and print an appropriate error message. The function should also handle any other exceptions that might occur, such as TypeError if the user inputs a non-string value?

```
# write your answer code here

```

Lesson Three : Finally Block

In any project after completing the tasks, it is recommended to destroy all the unwanted things that are created for the completion of the task. For example, if I have opened a database connection in my code and done some tasks related to the database. After the tasks are completed, it is recommended to close the connection. These clean-up activities are very important and should be handled in our code.

What is finally Block in Python?

`finally` is a keyword in python which can be used to write a finally block to do clean-up activities.

💡 *Why can't we use* ***`try except`*** *block for clean-up activities?*

- **`try` block:** There is no guarantee like every statement will be executed inside the try block. If an exception is raised at the second line of code in the try block at, then the remaining lines of code in that block will not execute. So, it is not recommended to write clean up code inside the try block.
- **`except` block:** There is no guarantee that an except block will execute. If there is no exception then except block won't be executed. Hence, it is also not recommended to write clean up code inside except block.

So, a separate block is required to write clean-up activities. This block of code should always execute irrespective of the exceptions. If no exception, then clean-up code should execute. If an exception is raised and is also handled , then also clean-up code should execute. If an exception is raised, but not handled, then also clean-up code should execute.

All the above requirements, which can't be achieved by try except block, can be achieved by finally block. The finally block will be executed always, irrespective of the exception raised or not, exception handled or not.

Case 1: If there is no exception, then finally block will execute.

Ex 14.13 (Finally block without occurrence of exception)

```
distance = 120
time = 2
fuel_consumed = 10
Power = True

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError as e:
  print("Exception Information:", e)
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Speed = 60.0 mph
Power = False, Distance = 0 miles, Time = 0 Hrs
```

Case 2: If an exception is raised and handled with except block, then also finally block will be executed.

Ex 14.14 (Finally block with occurrence of exception but handled)

```
distance = 120
time = 0
fuel_consumed = 10
Power = True

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError as e:
  print("Exception Information:", e)
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Exception Information: division by zero
Power = False, Distance = 0 miles, Time = 0 Hrs
```

Case 3: If an exception raised inside try block but except block doesn't handle that particular exception, then also finally block will be executed.

Ex 14.15 (Finally block with occurrence of exception but not handled)

```
distance = 120
time = 0
fuel_consumed = 10
Power = True

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ValueError as e:
  print("Exception Information:", e)
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Power = False, Distance = 0 miles, Time = 0 Hrs
---------------------------------------------------------
--------------------
ZeroDivisionError                         Traceback
(most recent call last)
<ipython-input-37-77da96561b94> in <cell line: 6>()
      5
      6 try:
----> 7   speed = distance / time
      8   print(f"Speed = {speed} mph")
      9 except ValueError:

ZeroDivisionError: division by zero
```

EXERCISE 14.5

Write a Python program that takes a series of numbers from user, and then calculates the average of the numbers. If the user enters a non-numeric value, the program should catch the ValueError and prompt the user to enter a valid number. Ensure that the program continues to take input from the user, even if an exception is raised, by using the finally clause?

```
# write your answer code here

```

Nested `try-except-finally` blocks:

In python nested try except finally blocks are allowed. We can take try-except-finally blocks inside try block. We can take try-except-finally blocks inside except block. We can take try-except-finally blocks inside finally block

Different cases and scenarios

Case 1: If no exception raised then outer try, inner try, inner finally, outer finally blocks will get executed.

Ex 14.16 (Nested blocks without exception raised in outer try)

```
distance = 120
time = 2
fuel_consumed = 10
Power = True

try:
  speed = distance / time
  print(f"Speed = {speed} mph")

  try:
    mileage = distance / fuel_consumed
    print(f"Mileage = {mileage} MPG")

  except ZeroDivisionError:
    print(f"Can't divide {distance} by {fuel_consumed}.")

except:
  print("Default Exception: Please provide valid input only!")

finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Speed = 60.0 mph
Mileage = 12.0 MPG
Power = False, Distance = 0 miles, Time = 0 Hrs
```

Case 2: If an exception is raised in an outer try, then outer except blocks are responsible to handle that exception.

Ex 14.17 (Nested blocks with exception raised in outer try)

```
distance = 120
time = 0
fuel_consumed = 10
Power = True
try:
  speed = distance / time
  print(f"Speed = {speed} mph")
  try:
    mileage = distance / fuel_consumed
    print(f"Mileage = {mileage} MPG")
  except ZeroDivisionError:
    print(f"Can't divide {distance} by {fuel_consumed}.")
except:
  print("Default Exception: Please provide valid input only!")
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Default Exception: Please provide valid input only!
Power = False, Distance = 0 miles, Time = 0 Hrs
```

Case 3: If an exception raised in inner try block then inner except block is responsible to handle, if it is unable to handle then outer except block is responsible to handle.

Ex 14.18 (Exception handled by inner `except` block)

```
distance = 120
time = 2
fuel_consumed = 0
Power = True
try:
  speed = distance / time
  print(f"Speed = {speed} mph")
  try:
    mileage = distance / fuel_consumed
    print(f"Mileage = {mileage} MPG")
  except ZeroDivisionError:
    print(f"Can't divide {distance} by {fuel_consumed}.")
except:
  print("Default Exception: Please provide valid input only!")
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Speed = 60.0 mph
Can't divide 120 by 0.
Power = False, Distance = 0 miles, Time = 0 Hrs
```

Ex 14.19 (Exception handled by outer except block)

```
distance = 120
time = 2
fuel_consumed = 0
Power = True
try:
  speed = distance / time
  print(f"Speed = {speed} mph")
  try:
    mileage = distance / fuel_consumed
    print(f"Mileage = {mileage} MPG")
  except ValueError:
    print(f"Can't divide {distance} by {fuel_consumed}.")
except:
  print("Default Exception: Please provide valid input only!")
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Speed = 60.0 mph
Default Exception: Please provide valid input only!
Power = False, Distance = 0 miles, Time = 0 Hrs
```

EXERCISE 14.6

Write a Python function that takes length & width of a rectangle as input to calculate area? The function should handle the following exceptions using nested try except block:

- `ValueError: If the user inputs non-numeric values for length or width.`
- `ValueError: If the user inputs negative values for length or width.`

```
# write your answer code here

```

💡 *Can we use `else` block in exception handling?*

`else` block in exception handling

Yes, we can use else blocks with try-except-finally blocks. The else block will be executed if and only if there are no exceptions inside the try block. If no exception then try, else and finally blocks will get executed.

💡 *What if you want to convert the speed from mph to km/hr?*

Ex 14.20 (`else` block without exception raised)

```
distance = 120
time = 2
fuel_consumed = 10
Power = True

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError:
  print(f"Can't divide {distance} by {fuel_consumed}.")
else:
  speed_in_kmph = speed * 1.60934
  print(f"Speed in kmph = {speed_in_kmph} km/hr")
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Speed = 60.0 mph
Speed in kmph = 96.5604 km/hr
Power = False, Distance = 0 miles, Time = 0 Hrs
```

If an exception is raised inside the try block, then the except block will get executed but the else block won't.

Ex 14.21 (`else` block with exception raised)

```
distance = 120
time = 0
fuel_consumed = 10
Power = True

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError:
  print(f"Can't divide {distance} by {fuel_consumed}.")
else:
  speed_in_kmph = speed * 1.60934
  print(f"Speed in kmph = {speed_in_kmph} km/hr")
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Can't divide 120 by 10.
Power = False, Distance = 0 miles, Time = 0 Hrs
```

Possible Combinations with `try-except-else-finally`

- Only `try` block is **invalid**, only `except` block is **invalid**, only `finally` block is **invalid**.
- `try` with `except` combination is valid → `try` block follows `except` block.
- `try` with `finally` combination is valid → `try` block follows `finally` block.
- `try`, `except` and `else` combination is valid → `try`, `except` blocks follows `else` block.

Ex 14.22 (SyntaxError of using only `try` block)

```
try:
  speed = distance / time
  print(f"Speed = {speed} mph")
```

```
  File "<ipython-input-60-76a5e2145726>", line 3
    print(f"Speed = {speed} mph")
                                  ^
SyntaxError: incomplete input
```

Ex 14.23 (SyntaxError of using only `except` block)

```
except:
  print("Default Exception: Please provide valid input only!")
```

```
  File "<ipython-input-61-11ff9cc50efa>", line 1
    except:
    ^
SyntaxError: invalid syntax
```

Ex 14.24 (SyntaxError of using only `finally` block)

```
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
  File "<ipython-input-62-b52137916ee6>", line 1
    finally:
    ^
SyntaxError: invalid syntax
```

Ex 14.25 (`try` with `except` combination)

```
distance = 120
time = 2
fuel_consumed = 10

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError:
  print(f"Can't divide {distance} by {fuel_consumed}.")
```

```
Speed = 60.0 mph
```

Ex 14.26 (`try` with `finally` combination)

```
distance = 120
time = 2
fuel_consumed = 10
Power = True

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
finally:
  # cleanup and reset everything
  Power = False
  distance = 0
  time = 0
  print(f"Power = {Power}, Distance = {distance} miles, Time = {time} Hrs")
```

```
Speed = 60.0 mph
Power = False, Distance = 0 miles, Time = 0 Hrs
```

Ex 14.27 (`try`, `except` and `else` combination)

```
distance = 120
time = 2
fuel_consumed = 10

try:
  speed = distance / time
  print(f"Speed = {speed} mph")
except ZeroDivisionError:
  print(f"Can't divide {distance} by {fuel_consumed}.")
else:
  speed_in_kmph = speed * 1.60934
  print(f"Speed in kmph = {speed_in_kmph} km/hr")
```

```
Speed = 60.0 mph
Speed in kmph = 96.5604 km/hr
```

EXERCISE 14.7

Write a Python function that takes an amount and a currency code as input, and returns the equivalent amount in a target currency?

Use a try-except-else block to handle the currency conversion:

- `try block: Attempt to retrieve the exchange rate for the given currency code from a dictionary.`
- `except block: Catch any KeyError caused by accessing a currency does not exist in the dictionary.`
- `else block: Execute code to perform the currency conversion and return the result.`

```
# write your answer code here

```

Lesson Four : Custom Exceptions in Python

Exceptions can be classified into two types based on how they are defined and used in the context of exception handling:

1. Custom Exceptions
2. Predefined Exceptions.

1. Predefined Exceptions in Python

The exceptions which we handled in the examples till now are predefined ones. They are already available builtin exceptions. If any exception occurs, then Python Virtual Machine will handle those exceptions using the predefined exceptions.

Examples of predefined exceptions are ZeroDivisionError, ValueError, NameError etc.

2. Custom Exceptions in Python:

Sometimes based on project requirement, a programmer needs to create his own exceptions and raise explicitly for corresponding scenarios. Such types of exceptions are called customized Exceptions or Programmatic Exceptions. We can raise a custom exception by using the keyword `raise`.

Steps to create Custom Exception in python:

The first step is to create a class for our exception. Since all exceptions are classes, the programmer is supposed to create his own exception as a class. The created class should be a child class of in-built "Exception" class.

> Syntax:
> ```
> class Exception_Name(exception):
>     def --init--():
>         self.arf = arg
> ```

In the second step raise the exception where it required. When the programmer suspects the possibility of exception, then he can raise his own exception using raise keyword

> Syntax:
> ```
> raise Exception_Name("message")
> ```

Ex 14.28 (Creating Custom Exception)

```
class NegativeError(Exception):
   def __init__(self, data):
       self.data = data
try:
   distance = int(input("Enter the distance: "))
   if distance < 0:
       raise NegativeError(distance)
except NegativeError as e:
   print(f"You provided {e}. Please provide positive integer values only")
```

```
Enter the distance: -100
You provided -100. Please provide positive integer values only
```

Ex 14.29 (Custom Exceptions)

```
class NegativeDistanceException(Exception):
   def __init__(self, message):
       self.msg = message

class NegativeTimeException(Exception):
   def __init__(self, message):
       self.msg = message

distance = int(input("Enter distance in miles:\n"))
time = int(input("Enter time in hours:\n"))

if distance < 0:
   raise NegativeDistanceException("The distance you provide is negative!")
elif time < 0:
   raise NegativeTimeException("The time you provide is negative!")
else:
   print("Please provide only valid inputs! and Try again!")
```

```
Enter distance in miles:
100
Enter time in hours:
-2
---------------------------------------------------------
--------------------
NegativeTimeException                     Traceback
(most recent call last)
<ipython-input-75-98792ac33b72> in <cell line: 12>()
     13     raise NegativeDistanceException("The
distance you provide is negative!")
     14 elif time < 0:
---> 15     raise NegativeTimeException("The time you
provide is negative!")
     16 else:
     17     print("Please provide only valid inputs! and
```

EXERCISE 14.8

Write a Python function that takes a password as input and validates it against the following rules:

- `The password must be at least 8 characters long.`
- `The password must contain at least one uppercase letter, one lowercase letter, and one digit.`

If the password is invalid, raise a custom PasswordError exception with an appropriate error message.

```
1 # write your answer code here
2
```

Key Takeaways

- *An error is an unexpected or exceptional condition that occurs during the execution of a program.*
- *There are two types of errors in programming; Syntax error and Runtime error.*
- *The error which occurs because of invalid syntax is called syntax error.*
- *Runtime error is an error that occurs during the execution of a program.*
- *Based on the success of execution, the program execution flow can be categorized into two types: Normal Flow and Abnormal Flow.*
- *An unwanted or unexpected event which disturbs the normal flow of the program is called exception.*
- *Errors indicate serious issues that cannot be fixed, while exceptions deal with specific problems that can be resolved.*
- *Exception handling is a way to deal with unexpected or exceptional situations that can occur while running a program.*

- *The `try-except` block is used to catch and handle exceptions.*
- *`finally` is a keyword which can be used to write a finally block to do clean-up activities.*
- *Exceptions can be divided into two types: Custom Exceptions and Predefined Exceptions.*
- *Predefined exceptions are exceptions that are already defined as part of the programming language or framework.*
- *Custom exceptions are programmer-defined exceptions used to handle specific exceptional conditions or errors.*

// END OF CHAPTER FOURTEEN

QUICK REFERENCE

A

- Abnormal flow refers to the termination of a program's normal execution due to the occurrence of a runtime error or exception, without completing the remaining statements.
- Abstraction is the process of simplifying complex things by focusing on the important parts and hiding unnecessary details.
- add() is a method used to add individual items to a set.
- Aliasing is the process of giving a new name to an existing one.
- append() is a method we can add elements to the list.
- Argument is a variable (which contains data) or a parameter which is sent to the function as input
- Arithmetic operators perform basic mathematical calculations such as +, -, x, and %.
- Assignment operators assign values to variables using the '=' symbol.

B

- Bitwise Operators are operators used to perform bitwise calculations on integers.
- break statement is a statement that terminates a loop.

C

- Class is like a blueprint or a template for creating objects.
- Class methods are methods which act upon the class variables or static variables of the class.
- Class variables are variables that are defined within a class.
- clear() is a method used to remove all the elements from a sequence-based data structure.
- Cloning is the process of creating duplicate independent objects.
- Comment is a programmer-readable explanation or annotation in the source code.
- Compiler is a program that translates code written in one programming language into a language that a computer can understand and execute.
- Compound operators combine arithmetic and assignment operators (+=, -=, =, /=, %=, *=, //=).
- Conditional Statements are statements executed based on the condition.
- Constructor is a special method within a class that is automatically called when an object is created.
- continue statement is a statement that skips the rest of the code for the current iteration.
- copy() is a method used to copy all the elements from one list to another.
- count() is a method to find the number of occurrences of element present in the sequence-based data structure.
- Custom exceptions are programmer-defined exceptions used to handle specific exceptional conditions or errors.

D

- Data Structure is a particular way of organizing data in a computer.
- Data Type is type of data that to be stored in a variable.
- Decorator is a special function which adds some extra functionality to an existing function.

- del is a statement used to delete an element at a given index.
- Dictionary comprehension is a concise way of creating dictionaries in Python using an iterable object.
- Dictionary is an unordered collection of items.
- difference_update() is method which updates the set calling difference_update() method with difference of sets.
- difference() is a method which returns the elements present in set A but not in set B.
- discard() is a method removes the specified element from the set. If the specified element is not present in the set, then we won't get any error.
- Duck typing is a programming style, in which the object passed to a method supports all the attributes expected from it.

E

- Encapsulation is a way to restrict the direct access to some components of an object.
- Error is an unexpected or exceptional condition that occurs during the execution of a program.
- eval() function takes the expression in the form of string and evaluates it and returns the result.
- Exception handling is a way to deal with unexpected or exceptional situations that can occur while running a program.
- Exception is unwanted or unexpected event which disturbs the normal flow of the program.
- extend() is a method used to append the elements from one list to the end of another list.

F

- finally is a keyword which can be used to write a finally block to do clean-up activities.
- find() is a method that returns an index of the first occurrence of the given substring.
- for loop is a control flow statement that allows you to execute a block of code a specific number of times.
- Frozen set is similar to a set, but it is immutable, meaning its elements cannot be changed after creation.
- Function is a group of statements or a block of code to perform a certain task.
- Functional oriented programming is a program paradigm where programs are built around functions.

G

- Garbage collection is an automatic process that manages the memory occupied by objects
- Generators are just like functions which give us a sequence of values one as an iterable.
- get() is a method used to get the value associated with the key in a dictionary.
- Getter methods are methods which are used to get values of instance variables.

I

- IDE is a graphic interface to write, run, debug and also convert code to machine code.
- Identity operators compare the memory location (address) of two variables to check for identity.
- Indentation is the spaces at the beginning of a code line to indicate a block of code.
- index() is a method to find a substring in forward direction.
- Indexing refers to the position of characters within a string.
- Infinite loop is a loop that never ends because its condition never becomes false.
- Inheritance is the process of creating new classes from already existing classes.

- input() is a function that can accept values from a user (you).
- insert() is a method used to add the elements to the list at the specified index (position).
- Interface is an abstract class which contains only abstract methods and doesn't contain any concrete methods.
- Interpreter is a program that directly executes code written in a programming language without prior translation into a machine-readable form.
- intersection_update() is a method which updates the set calling intersection_update() method with the intersection of sets.
- intersection() is a method which returns common elements present in both set A and set B.
- isdisjoint() is a method which returns True if two sets are disjoint sets. If not, it returns False.
- issubset() is a method which returns True if all elements of a set are present in another set. If not, it returns False.
- issuperset() is a method which returns True if a set has every elements of another set. If not, it returns False.
- items() is a method which returns list of tuples representing key-value pairs.

J

- join() is a method used to join a group of strings (list or tuple) with respect to the given separator.
- Jupyter Notebook is a popular IDE for data science and machine learning tasks.

K

- keys() is a method which returns all keys.
- Keywords are special reserved words, that have specific meanings and purposes.

L

- Lambda function is a short anonymous function that can be defined inline without a name.
- len() is a method used to get the length or the number of items in a given sequence.
- List comprehension is a concise way of creating lists using iterable objects like tuples, strings, and lists.
- List is a data structure which can store a group of elements or objects in a single variable.
- Local variables are variable which we declare inside of the method.
- Logical Operators are operators which do some logical operation on the operands and return True or False.
- Loop is a repetitive construction in programming.

M

- max() is a method used to find the maximum value from the given sequence.
- Membership operators are operators which are used to check whether an element is present in a sequence of elements or not.
- Method is a function that is associated with a specific object or class.
- min() is a method used to find the minimum value from the given sequence.

N

- Name mangling is a technique used to create "protected" or "private" attributes within a class.
- Nested class is a class inside another class.

- Nested function is a function that is defined inside another function.
- Nested if statement is an if statement inside another if statement.
- Nested list is a list within another list.
- Nested loop is a loop that is written inside another loop.
- Normal flow refers to the successful execution of a program's statements without any interruptions or errors, resulting in the expected output.

O

- Object is instance of a class.
- Object-oriented programming (OOP) is a way of organizing and designing computer programs.
- Operator is a symbol used to perform actions or operations on variables.
- Overloading is the ability to define multiple methods in a class with the same name but different parameters.
- Overriding is the process of redefining a method in a child class that already exists in the parent class.

P

- pass statement is a statement used as a placeholder.
- Polymorphism is the ability of an object to take on many forms.
- pop() is a method used to remove and return an item from a list, set, or dictionary.
- popitem() is a method used to remove an arbitrary item(key-value) from the dictionary and returns it.
- Predefined exceptions are exceptions that are already defined as part of the programming language or framework.
- print() is a function that can print the specified message to your computer screen.
- Procedural programming is like following a step-by-step procedure or a set of instructions.
- Programming is the process of preparing an error-free program for a device.
- Programming language is a formal language to communicate with a computer through instructions.
- Python is a high-level programming language known for its simplicity and readability.

R

- random.choice() is a method used to randomly select and return an item from a sequence.
- Recursive function is a function when it is called by itself.
- Relational operators check relations between operands, such as greater than or less than.
- remove() is a method used to randomly select and return an item from a sequence.
- replace() is a method used to replace old string with new string.
- reverse() is a method which reverses the order of list elements.
- Runtime error is an error that occurs during the execution of a program.

S

- self is a default variable that refers to a current class object.
- Set is a collection of unique elements.
- setdefault() is a method which returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.
- Setter methods are methods which are used to set values of instance variables.

- Slicing refers to extracting a substring from a string.
- sort() is a method used to rearrange the elements of a list in a specific order, typically the default natural sorting order.
- split() is a method used to split the given string according to the specified separator.
- Static methods are class-level methods that are not bound to specific instances of a class.
- Static variable is a variable that is associated with a class, not with a specific instance of that class.
- String is a sequence of characters.
- strip() is a predefined methods to remove spaces in Python.
- sum() is a method used to calculate the total or sum of all the elements in a sequence.
- super() function is a built-in function that allows you to call a method in a parent from a child class.
- symmetric_difference_update() is a method which finds the symmetric difference of two sets and updates the set calling it.
- symmetric_difference() is a method which returns elements present in either set A or set B but not in both.
- Syntax error is an error which occurs because of invalid syntax.
- Syntax is set of rules you must follow to run your program correctly.

T

- try-except block is used to catch and handle exceptions.
- Tuple is a collection of objects, which are ordered and can’t be changed.
- type() is a function, which is used to check data type of a variable.

U

- union() is a method which returns all elements present in both set A and set B.
- update() is a method used to add multiple items to a set.

V

- values() is a method which returns all values of a dictionary.
- Variable is the name which we give to the memory location which holds some data.

W

- while loop is a control flow statement that allows you to execute a block of code repeatedly as long as a certain condition is true.

No content here - this page is intentionally blank.

FUN-DAMENTALS OF PYTHON PROGRAMMING

Gateway for the Joy of Programming

To better communicate with a computer you should have to learn a programming language, Even though you have translators like compiler and interpreter. But which programming language? The answer would be simple, used for different / multiple purposes, i.e. Python.

Python is a programming language that does it all, from web applications to video-games, Data Science, Machine Learning, real-time applications to embedded applications, and so much more. In this lesson, we'll take a deeper dive into a broader list of applications of Python out in the wild.

This book is for everyone – from complete newbies to experienced programmers looking for a refresher. By the time you reach the end, you'll be writing your own programs, solving puzzles, and get ready to use other python libraries and frameworks. Most importantly, you'll rediscover the joy of learning and the thrill of bringing your ideas to life through code.

Written by Bemnet Girma

Digitally Published Book