

Anilkumar Para

# Python

Guido van Rossum

# Why you should learn a programming language?

- To make use of the computer power
- To improve your Abstract thinking and problem solving skills

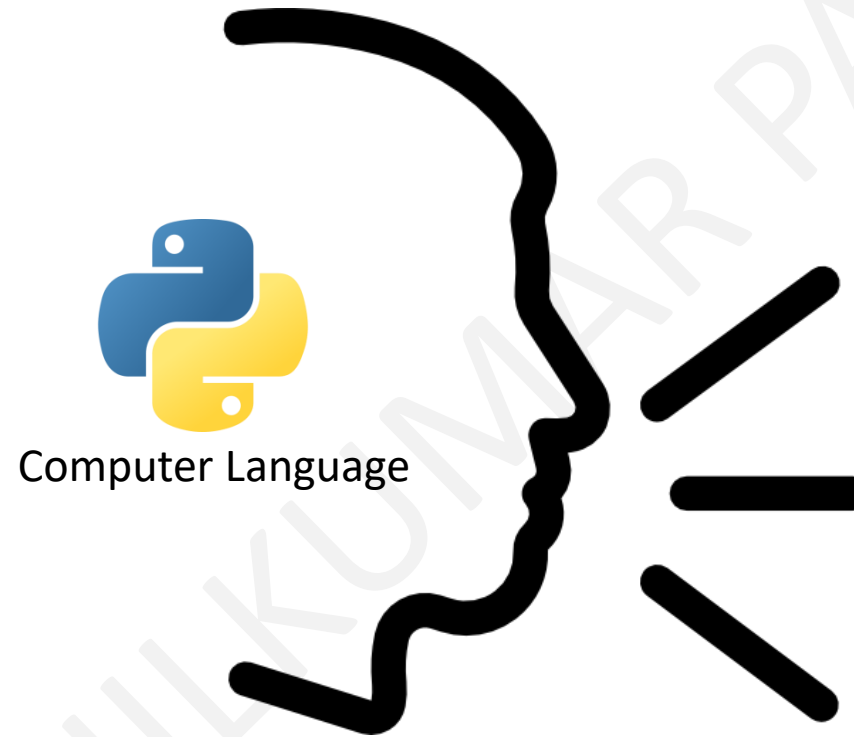
# Why you should learn a programming language?

- To make use of the computer power
  - Let's say if you want multiply to  $20901892523 * 345718930647$ , Human may need around 5 minutes to calculate, but computer does in ms. So, it has the computational power (CPU: Processor) and memory to store the data
  - Key points are computer is faster than human and doesn't make any mistakes
  - On day to day basis you use the google, Facebook, WhatsApp, Instagram, those apps are developed using one of the programming language
  - Learning a programming language will give the ability to develop the software applications
  - If you have these skills you can grab any Software job opportunities
- To improve your Abstract thinking and problem-solving skills
  - When you write a code, you are going to think logic of certain problem and indirectly you are going to solve the problem
  - Give the example of the prime number
  - Tell that later onwards computer will just do in ms , but human has to determine every time

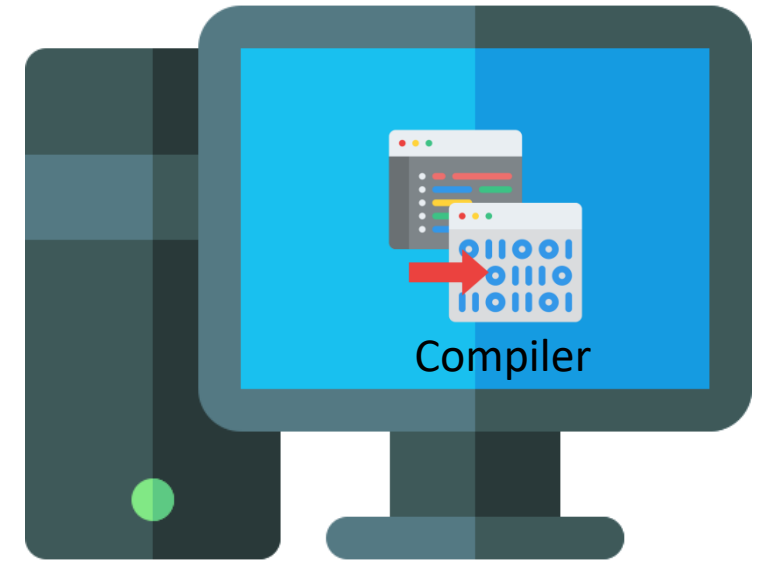
# Why you should learn a programming language?



**Human**



In order to speak to computer, human should learn the computer language



**Computer**

# Why you should learn a programming language?

- In order to communicate and make use of the computer computational power and memory you should learn a programming language. If you want to use the computer you don't need any programming language, just plain English is sufficient
- Compilers understand the programming language and they are mediators between the programmer and machine.
  - You write simple program, it compiles and converts to machine code and machine perform the operations and returns the output in the form of human readable format
  - To connect this point to real-world example, let's say if you want to communicate with different parts of people you should have a common medium Hindi/English
  - You have a necessity with the computer so, you need to learn the programming language which computer understands
  - Show simple program

## Famous Quote

- “Everybody in this world should learn how to program a computer.. Because it teaches you how to think”

- Steve Jobs

# Why Python?

- Python uses the simple English and mathematics
- Whoever knows English and Mathematics they can learn python
- Python Code is as understandable as plain English
- **code readability**, its **syntax allows programmers to express concepts in fewer lines of code.**

## Examples for code readability

- `a=5` # write similarly how we write in the maths
- `print(a)` => simply like an English
- Block of code spaces like English paragraphs



## Examples for code syntax

- Declare `a,b,c=1,2,3` in Python and not possible in C
- Swapping of 2 numbers
- Find the last element in array/list
- Finding the square of each number in a list/array

## Reasons for increasing popularity

- Emphasis on **code readability** (human readable example indentation , in English we don't use {}), **shorter codes**, ease of writing.
- Programmers can express **logical concepts in fewer lines of code** in comparison to languages such as C or C++ or Java.

## Reasons for increasing popularity

- Python supports multiple programming paradigms, like **object-oriented**, imperative and functional programming or **procedural**.
- There exists **inbuilt functions** for almost all the frequently used concepts. It has around **1,37,000 libraries**
- Philosophy is “**Simplicity** is the best”.

# Applications of Python

- Several **Linux distributions use installers** written in **Python**.
- **Example:** In Ubuntu we have the Ubiquity.
- **Raspberry Pi** – Single board computer uses **Python** as its **principal user-programming language**.

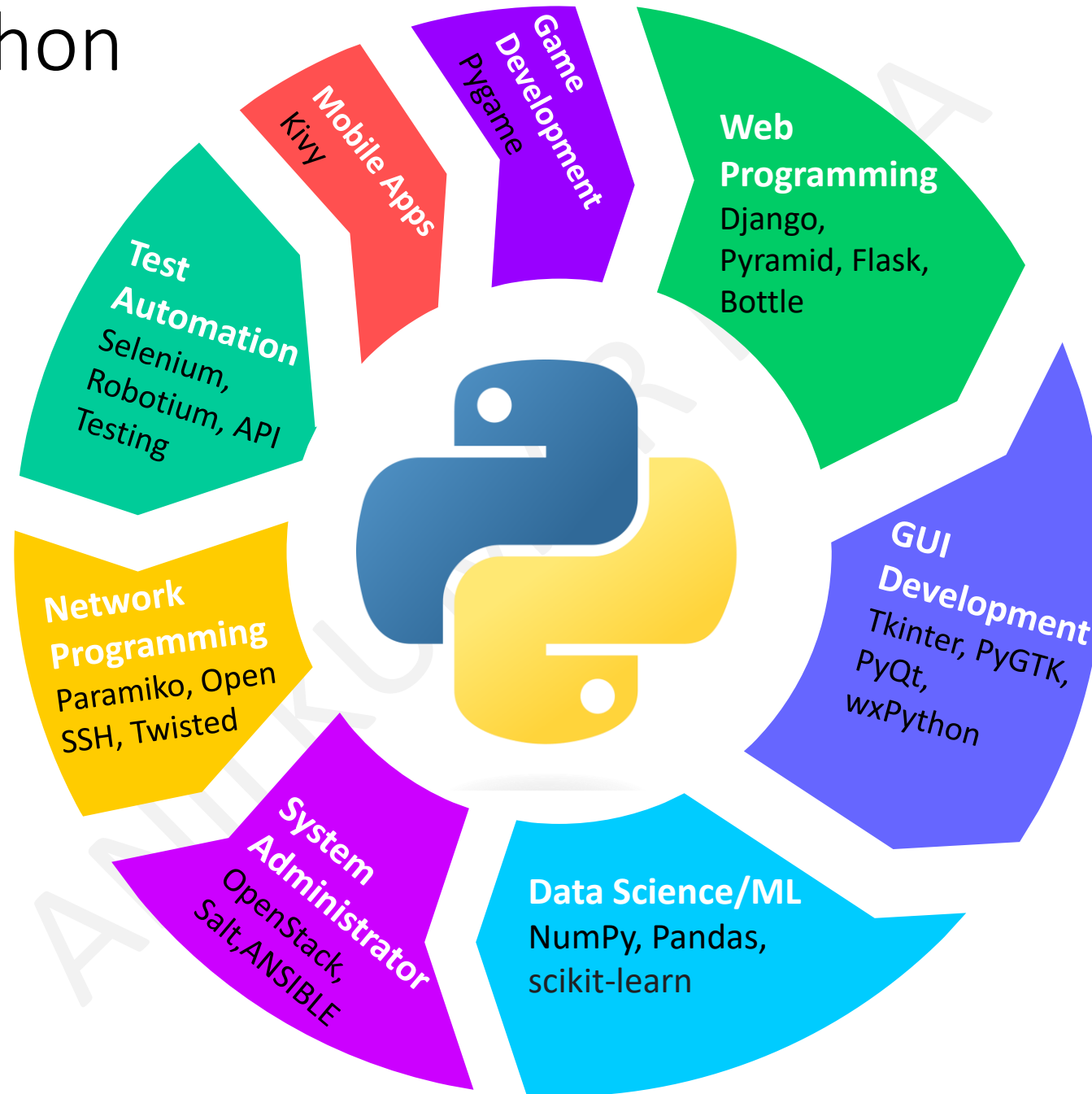
# Applications of Python

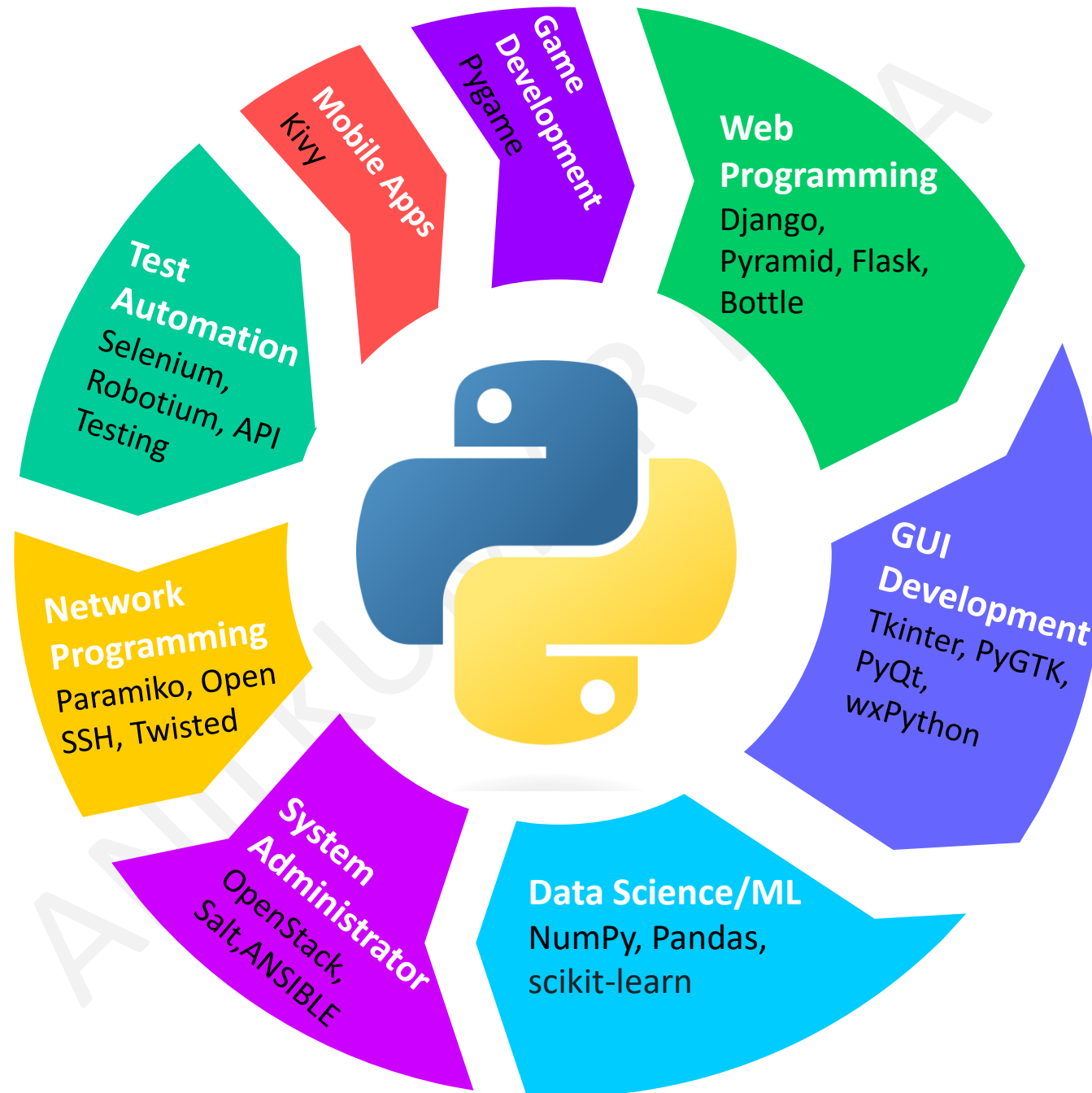
- Python is now being used in **Game development** areas.
- Python has seen extensive use in the **information security industry**, including in exploit development.

# Applications of Python

- Some of the games made with the Python
  - Battlefield 2
  - Civilization IV
  - EVE Online
  - Mount & Blade
  - Vampire: The Masquerade – Bloodlines
  - Snake
  - Pac-Man
  - Tic-tac-toe

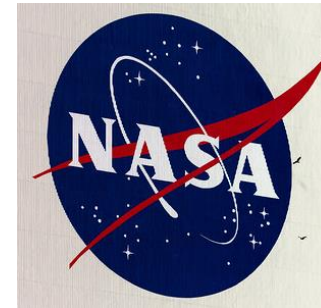
# Usage of Python







# Famous companies that uses Python



## Advantage of Learning Python

- As Python is used in many of the domains and applications. So, the Career Opportunities and Salaries are high.

## General opinion on Python

- If you don't know any programming language  
Python is easy to learn. But if you know any  
other programming language before learning  
Python you may feel little difficult for few days

## Quote

- You don't have to be genius to do the programming



# Intro

- Python was initially designed by **Guido van Rossum** in **1991** and developed by **Python Software Foundation**.
- From where Python name has come?
  - ❖ When Guido van Rossum began implementing Python, Guido van Rossum was also reading the published scripts from “Monty Python’s Flying Circus”, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language **Python**.





# Intro

- Python was initially designed by **Guido van Rossum** in **1991** and developed by **Python Software Foundation**.
- It was mainly developed for emphasis on **code readability**, its **syntax allows programmers to express concepts in fewer lines of code**.
- Python is an **open-source**
- Python is a programming language that lets you **work quickly and integrate systems more efficiently**.

# Intro (Contd.)

- Guido van Rossum began working on Python in the late 1980s as a successor to the ABC programming language and first released it in 1991 as Python 0.9.0.
- Python 2.0 was released in 2000 and introduced new features such as list comprehensions, cycle-detecting garbage collection, reference counting, and Unicode support.
- Python 3.0, released in 2008, was a major revision that is not completely backward-compatible with earlier versions.
- Python 2 was discontinued with version 2.7.18 in 2020

## Intro (Contd.)

- Latest stable version is **Python 3.11.3**
- [Python 3.12 \(in development\)](#)
- **Note:** Python 3.9+ cannot be used on Windows 7 or earlier.
- Python is a **high-level**, general-purpose programming language.



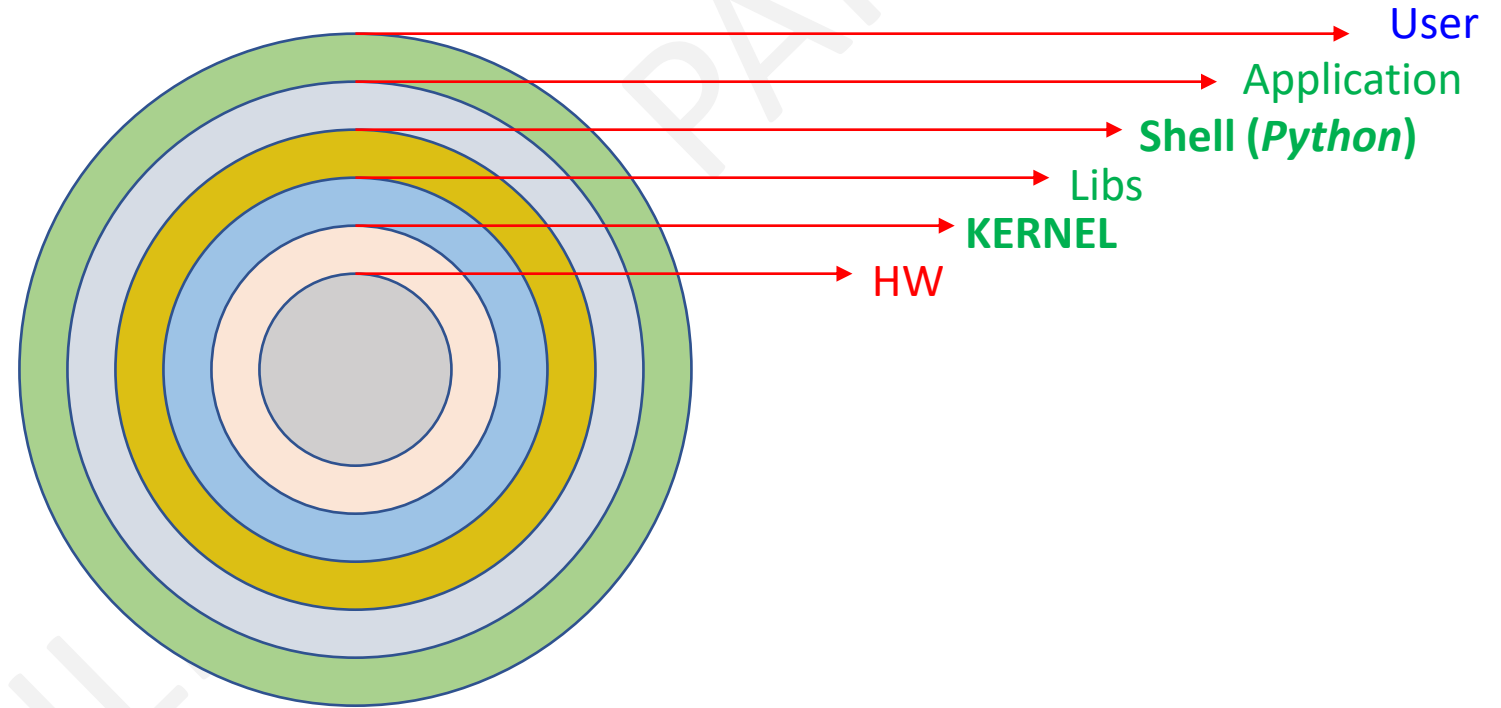
## Intro (Contd.)

- There are different variations of Python
- **Cpython:** Python written in C
- **CPython** is the **original Python implementation.** It is the implementation you download from Python.org. People call it CPython to distinguish it from other, later, Python implementations, and to distinguish the implementation of the language engine from the Python programming language itself.
- CPython is one of the many Python runtimes, maintained and written by different teams of developers.

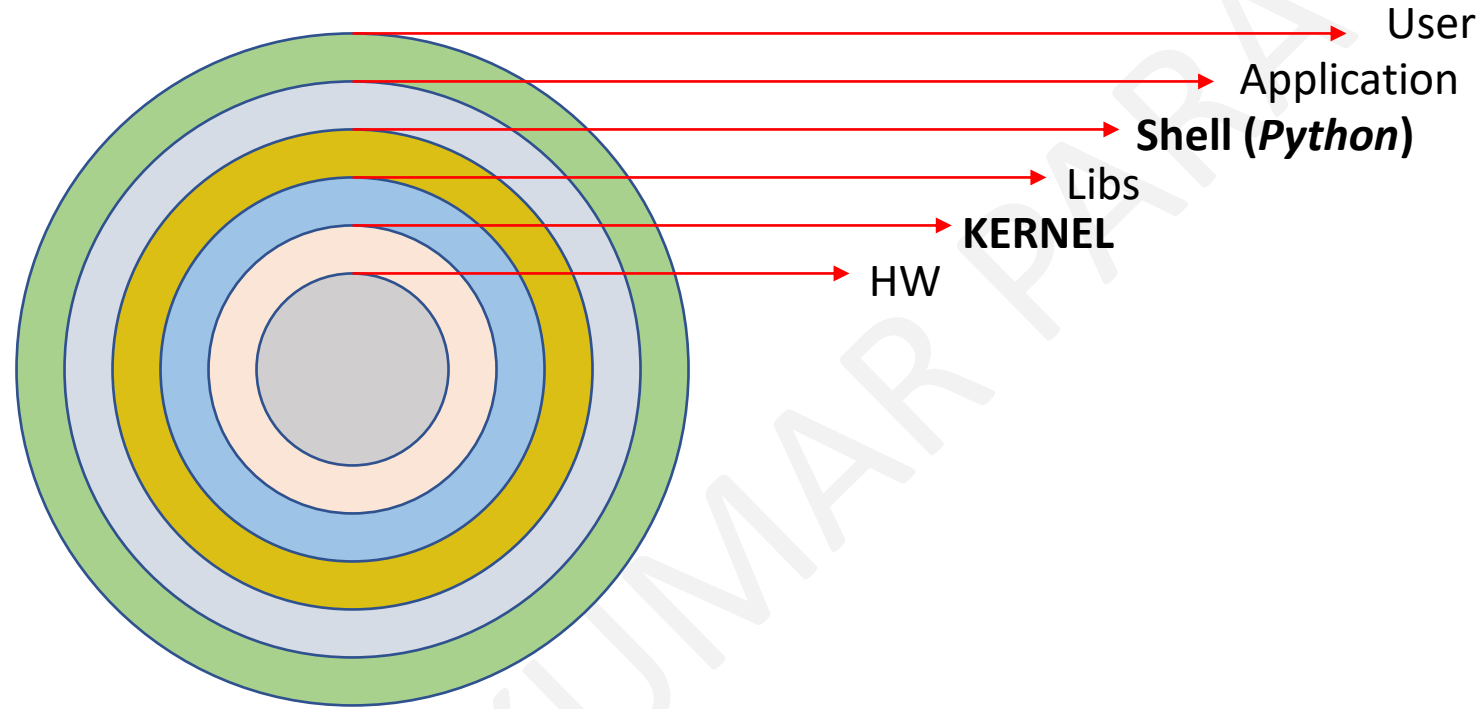
## Intro (Contd.)

- Some other runtimes you may have heard are PyPy, Cython, and Jython.
- **PyPy** - Python written in Python, includes a tracing JIT compiler
- Cython - Cython is a popular superset of Python. designed to give C-like performance with code that is written mostly in Python with optional additional C-inspired syntax.
- **Jython** - Python written in Java for the Java platform

# OS Architecture



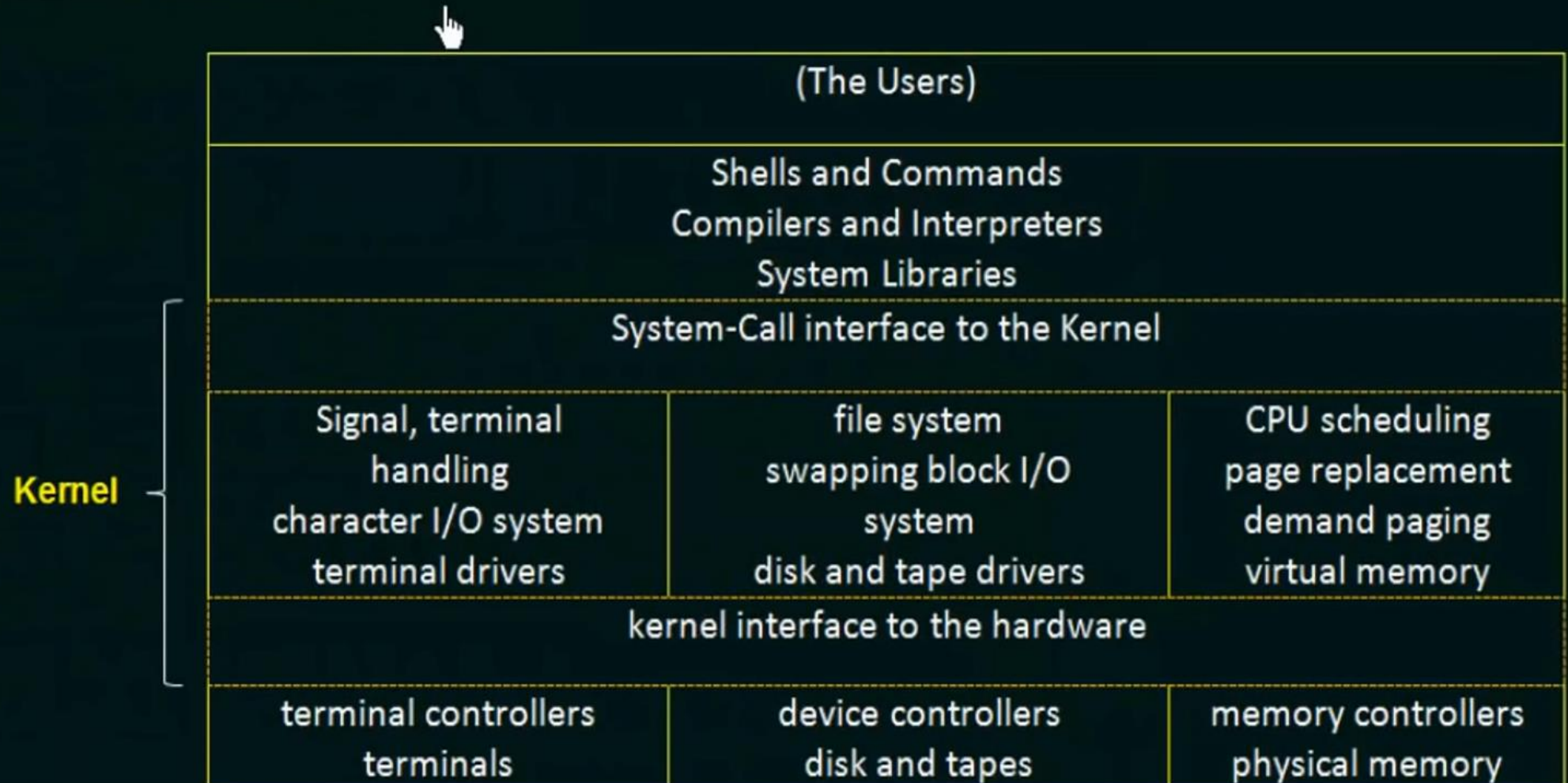
# OS Architecture



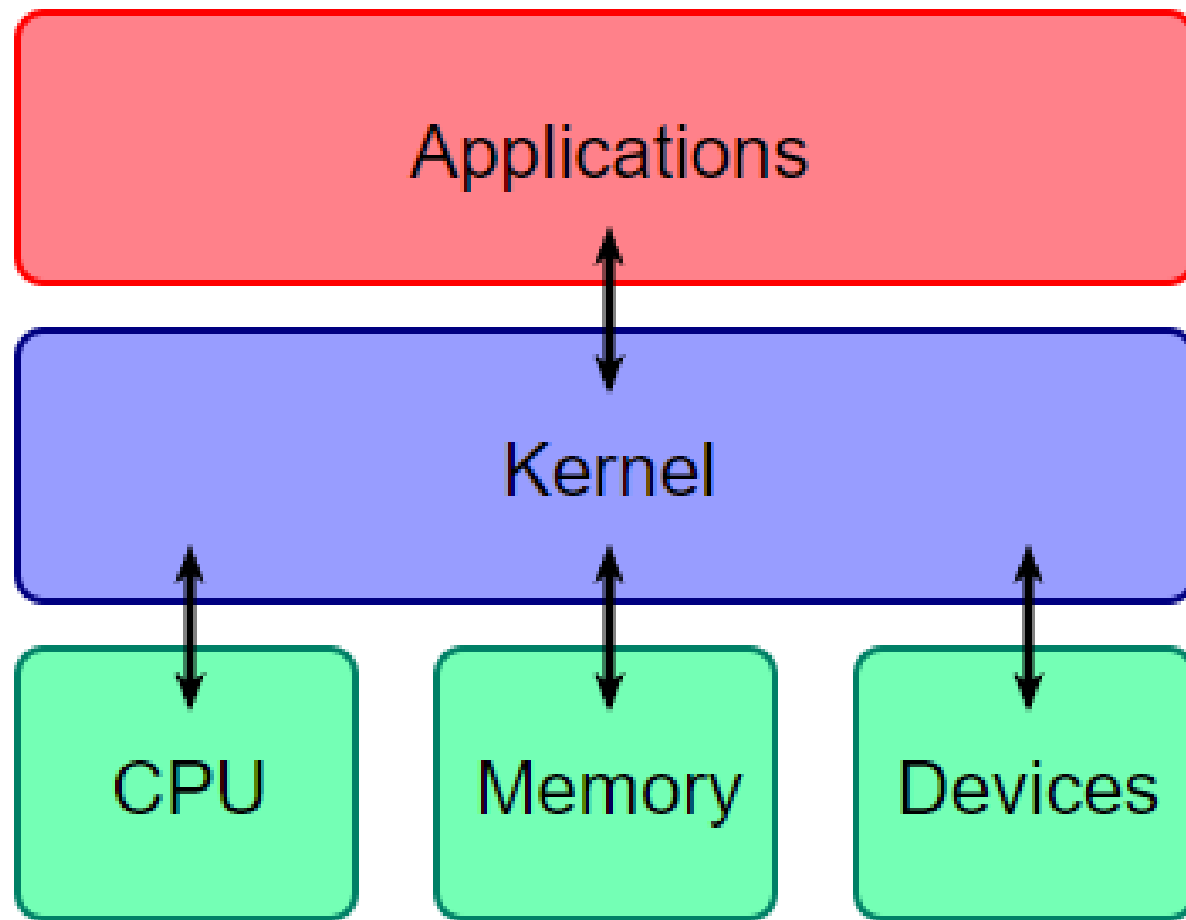
A **shell** is a computer program which exposes an [operating system](#)'s services to a user or other programs. The shell is **the layer of programming that understands and executes the commands a user enters**. In some systems, the shell is called a command interpreter.

# OS Structure

## Monolithic Structure



# Kernel



A kernel connects the [application software](#) to the hardware of a computer

# Low vs High level language

## Low level language

- A low-level language, often known as a **computer's native language**, is a sort of programming language. It is very close to writing **actual machine instructions**, and it deals with a computer's hardware components and constraints.
- Low-level code is **not human-readable**, and it is often cryptic.
- **Assembly language** and **machine language** are two examples of low-level programming languages.

## High level language

- High level language that resembles **natural language** or mathematical notation and is designed to reflect the requirements of a problem
- High level language is a language which can be easily **understandable by the human**
- **C**, **Java**, **Python** are examples of High-level language

## Low vs High level language (Platform Dependency)

### Low level language

- Low Level programming languages are **platform dependent** that means programs written in Low Level language can run on the same hardware with same configuration, you cannot run them on hardware that has different configuration.

### High level language

- High Level programming languages are **platform independent** that means programs written in High Level language can run on different hardware with different configuration.

**Platform** means Computer i.e. Computer and Software (Computer configuration).



## Low vs High level language (Speed )

### Low level language

- Low Level language programs are **faster** than High Level language programs as they do not need to convert.
- They have a **smaller number of syntaxes**, functions, keywords, class libraries.

### High level language

- High Level language programs are **slower** than Low Level language programs as they need to convert.
- They have a **greater number of syntaxes**, functions, keywords, class libraries.

## Low vs High level language (Easiness )

### Low level language

- Low Level language programs are **not as easy** as High-Level language. There are only two Low Level programming languages Binary and Assembly. Binary has only 0's, 1's, while Assembly has some **difficult type symbols** which are known as **mnemonics**.

### High level language

- Whereas the High-Level language programs are **easy** to write, read, modify and understand.

## Low vs High level language (Performance )

### Low level language

- Since, Low Level Languages programs are faster, so **performance** of Low-Level languages programs are **better** than the High-Level languages programs.

### High level language

- Since, High Level Languages programs are slower, so **performance** of High-Level languages programs are **little worse** than the Low-Level languages programs.

# Low vs High level language (Translation )

## Low level language

- Machine language will be in the binary, so **no need to translate** as Binary codes are Machine codes and computer understands them without any translations.
- Assembly language needs an Assembler to translate an Assembly program to its equivalent Binary/Machine code.

## High level language

- High Level Languages are **translated** by the compilers or interpreters; sometimes (in case of some programming languages) both compilers & interpreters are required to get the Object/Binary file.

## Low vs High level language (Flexibilities )

### Low level language

- Low level languages has little or **no libraries**

### High level language

- High Level languages have **huge libraries** with a rich set of Data types, keywords, functions etc. so these languages are good to develop an application with many great features using **less effort and resource.**

## Low vs High level language (Support )

### Low level language

- Low Level languages have **less support** than High Level Languages. There may be lesser number of professionals (community) in support of Low-Level languages as comparisons to High Level Language support.

### High level language

- Whereas high-level languages has **large developer community**, you **easily get a support**

## Interpreter

- Translates program **one statement** at a time.
- Compilation and execution take place **simultaneously**.
- It takes **less amount of time** to analyze the source code.

## Compiler

- Scans the **entire program** and translates it into machine code.
- The compilation is done **before** execution.
- It takes **large amount of time** to analyze the source code.

## Interpreter

- Overall execution time is **slower**.
- No Intermediate object code is generated. Hence are **memory efficient**.
- Continues translating the program until the first error is met, in which case it stops. Hence **debugging is easy**.

## Compiler

- Overall execution time is comparatively **faster**.
- Generates intermediate object code which further requires linking, hence **requires more memory**.
- It generates the error message only after scanning the whole program. Hence **debugging is comparatively hard**.



## Interpreter

- **Do not generate output program.** So, they evaluate the source program at every time during execution.
- Program Execution is a **part of Interpretation process**, so it is performed line by line.
- Always **source code is need** to run the program

## Compiler

- **Generates output program** (in the form of exe) which can be run independently from the original program.
- Program execution **is separate from the compilation**. It performed only after the entire output program is compiled.
- **No need of Source code**, you can run the program with .exe file

# Python Installation steps for Windows

1. Go to <https://www.python.org/>
2. Select the version of Python and download '.exe' file
3. Run Executable Installer.
4. Go to command prompt and verify Python version using (python --version) command
5. Verify Pip Was Installed (pip --version)
6. Write a simple program,  
to confirm the Python Installation

# PyCharm Installation

1. Go to <https://www.jetbrains.com/pycharm/download>
2. Select the community version
3. Run Executable Installer.
4. Configure the Interpreter
5. Write a simple program,  
to confirm the PyCharm Installation

# IDE



☐ Integrated development environment (IDE)

☐ Designed for **Developers**

# IDLE



☐ Integrated development and **learning** environment

☐ Designed for **Beginners (learners)**

# IDE



☐ It needs to be installed separately.

☐ Code Editor

☐ Syntax highlighting

# IDLE



☐ It comes with the Python installation.

☐ Code Editor

☐ Syntax highlighting

# IDE



☐ Code completion [Intelligent code completion]

☐ Debugging

☐ Refactoring, Code search

☐ Version control, Multiple language support

# IDLE

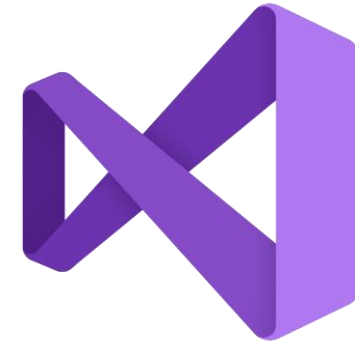


☐ Code completion [Auto code completion]

☐ Debugging



# IDEs



# print

For compilation Python takes only 49 MB and it can be installed on any processing system. Like raspberry Pi is credit card size hardware and there it will have only 100 MB RAM . There also you can run the Python coding and scripting. Raspberry PI is the key for the android and hardware testing. Again for testing we need Python. In Raspberry PI devices if you are not clearing or flushing the memory, system will be slow. So, if you want to do processing faster you need to clear the memory. In those cases you can use the Flush =True it clear the RAM and computer performance will be better. if you want the execution faster, you can use Flush=True. It shows the difference for only the low memory devices and high end devices you don't see any difference

Flush=True

This will be useful when you low memory (RAM) in the computer. It clears the data and re-uses the memory for the program. When you have good system configuration, you don't see any difference



# Print

- For compilation Python takes only 49 MB and it can be installed on any processing system. Like raspberry Pi is credit card size hardware and there it will have only 100 MB RAM . There also you can run the Python coding and scripting. Raspberry PI is the key for the android and hardware testing. Again for testing we need Python. In Raspberry PI devices if you are not clearing or flushing the memory, system will be slow. So, if you want to do processing faster you need to clear the memory. In those cases you can use the `Flush = True` it clear the RAM and computer performance will be better.
- if you want the execution faster, you can use `Flush=True`. It shows the difference for only the low memory devices and high end devices you don't see any difference  
`Flush=True`
- This will be useful when you low memory (RAM) in the computer. It clears the data and re-uses the memory for the program. When you have good system configuration, you don't see any difference

# Standard I/O functions

## Input

- input function. It is a Python inbuilt function
- It reads an input from standard input device
- Examples of input devices: Keyboard, mouse, Pen tablet, etc.

## Output

- print function. It is a Python inbuilt function
- It writes the output to standard output device
- Any display in my case it is laptop screen and if you are using the CPU, it will be a monitor, printer, etc.

# Keywords or Reserved words

- **Keywords** are the **reserved words** in Python.
- We **cannot use a keyword as variable name, function name or any other identifier.**
- **Keywords** have a special meaning in a language and are part of the syntax.

# Keywords or Reserved words

- `import keyword`
- `keyword.kwlist` => Gives the list of keywords available in the Python

# Variables or Identifiers

- Variable is a named memory location which will have value/s
- Variable is the name given to identify memory location

# Variables or Identifiers

- Unlike other programming languages, Python has no command for declaring a variable. Variables need not be declared first in Python
- A variable is created the moment you first assign a value to it

# Rules for creating Variables

- A variable name must start with a letter or the underscore(\_) character
- A variable name cannot start with a number
- You can't use the keywords as variable names

# Rules for creating Variables

- A variable name can only contain **alpha-numeric** characters and **underscores** (A-z, 0-9, and \_)
- Why we can't use other special characters in variable names is they are used as part of the operators
  - =, -, \*, /, %, //
  - >, <, ==, !=, >=, <=
  - |, &, !, and, or, not
  - >>, <<, |, &, !, ^
  - @ - decorator
  - \$, [], (), {} - meta chars - Regex
  - . For decimal and , separator



# Rules for creating Variables

- Variable names are **case-sensitive** (age, Age and AGE are three different variables)

—

- `_` is used to reference a last unreferenced value in the Python

# Rules for creating Constants

- We must use only capital letters for denoting constants.
- All the rules are same except that we can't use lower case alphabets for constants. It is just a programming standard
- It indicate that it is a constant and programmers shouldn't change the value
- ex:  $\text{PI} = 3.14$
- $\text{GRAVITY} = 9.8$

# Memory

## Human



- ☐ Short-term memory (Hippocampus [temporal lobe])
- ☐ Long-term memory (Neocortex)

## Computer

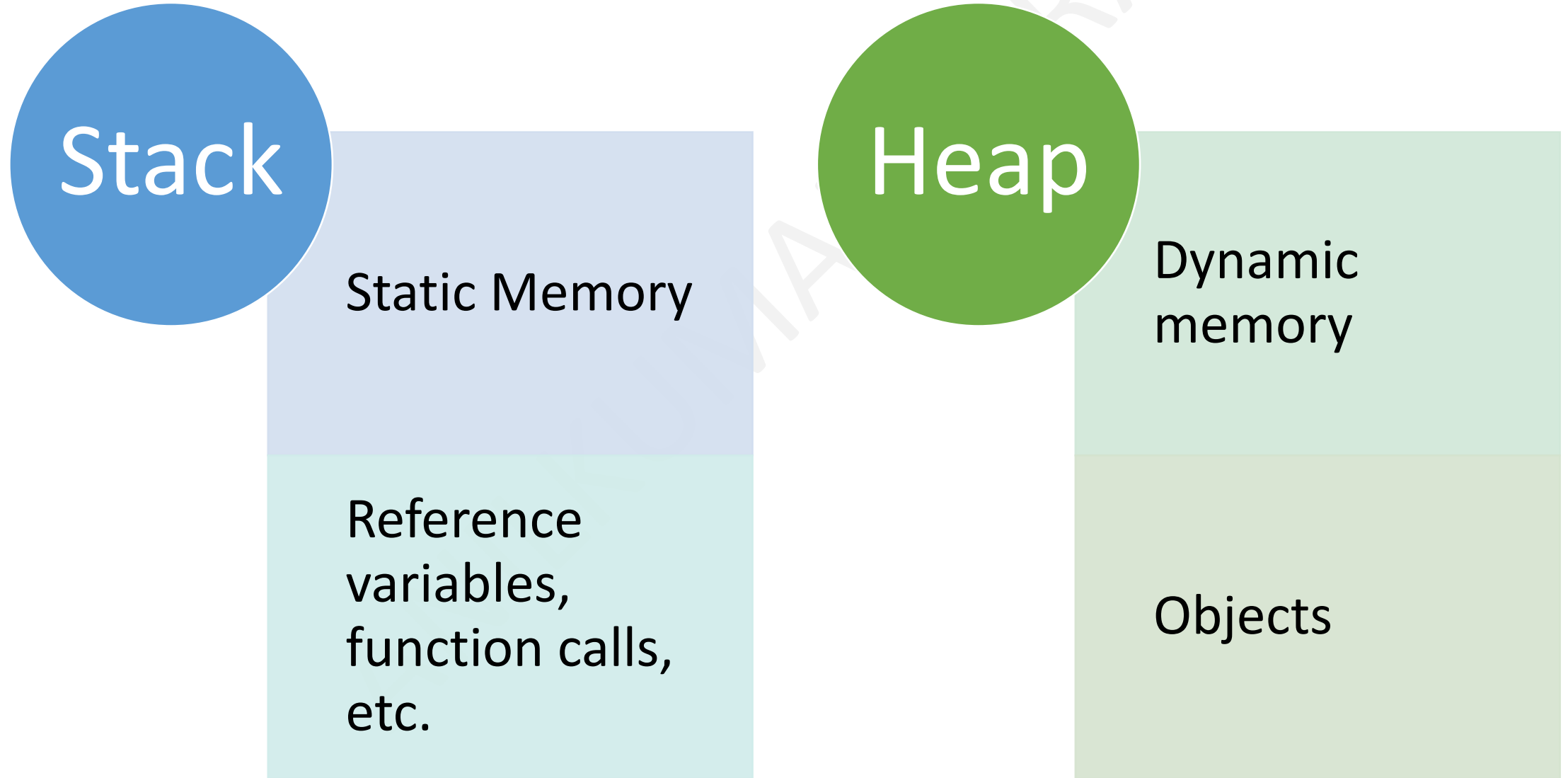


- ☐ Random Access Memory (RAM) => volatile
- ☐ Read-only memory (ROM) => non-volatile

# RAM

- Used for all computer tasks (opening a YouTube, opening a PowerPoint and running program, etc.)

RAM



# Points for memory allocation

- Assigning same value to different variables
- Memory management
- Private heap
- PMM delegates some of the work to specific locators
- Python heap managed by interpreter
- Cpython (malloc, calloc, realloc, free)
- Garbage collector & Reference counter

# Data Types

- **Data type** specifies what **type of data** it is.
- Data type is also used to **convert the data from one data type to another (type Casting)**.



# Data Types

- Following are the data types in Python
  - ❖ Numeric or Python Numbers
  - ❖ bool
  - ❖ Grouped [Strings, Arrays (Collections)]

# Numeric

- **Int**, **float**, **complex**

- **Int:**

- whole-valued (without fraction or decimal) positive or negative number or 0.

- **Syntax:**

- `x = 1    # int`

# Numeric

- **Float:**

- Float is used to represent real numbers and is written with a decimal point (dividing the integer and fractional parts.)

- **Syntax:**

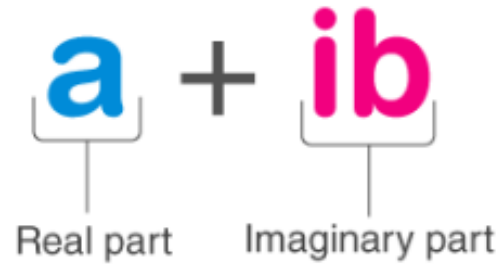
- `y = 2.8 # float`

- **Note:** Explain type function

# Numeric (Complex)

- **Complex:**
- The complex number is basically the combination of a real number and an imaginary number. The complex number is in the form of  $a+ib$ , where  $a$  = real number and  $ib$  = imaginary number.

A complex number is a number with 2 parts ;



The diagram shows the expression  $a + ib$  in a large font. The letter  $a$  is blue and the letters  $ib$  are pink. A black plus sign is between them. Below the  $a$ , there is a bracket and the text "Real part". Below the  $ib$ , there is a bracket and the text "Imaginary part".

$$\underbrace{a}_{\text{Real part}} + \underbrace{ib}_{\text{Imaginary part}}$$

# Numeric (Complex)

- Mostly we don't use complex numbers in test automation, python development

Complex numbers are used in **electrical** engineering all the time, because Fourier transforms are used in understanding oscillations that occur both in alternating current and in signals modulated by **electromagnetic** waves.

# Numeric (Complex) Programmatically

- Create a complex number from a real part and an optional imaginary part.
- **imag**: the imaginary part of a complex number
- **Real**: the real part of a complex number
- **conjugate(...)** method :
- Return the complex conjugate of its argument.  $(3-4j).conjugate() == 3+4j$ .

# Numeric (Complex)

- Complex:
- A complex number is created from two real numbers. Python complex number can be created using [complex\(\)](#) function as well as using direct assignment statement.
- Complex numbers support mathematical calculations such as addition, subtraction, multiplication and division.
- **Syntax:**
- Real+imagj or Real+imagJ
- Complex class has two attributes (real, imag) and one function(conjugate)

# Numeric (Complex)

- The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.
- The 'j' suffix comes from electrical engineering, where the variable 'i' is usually used for current.
- Complex numbers don't support comparison operators. If we try to execute `c < c1` then the error message will be thrown as `TypeError: '<' not supported between instances of 'complex' and 'complex'.`
- Python ~~cmath~~ `math` module provides access to mathematical functions for complex numbers.
- Explain about the help function



# Bool

**Python boolean** type is one of the built-in data types provided by Python, which represents one of the two values i.e., **True** or **False**. Generally, it is used to represent the truth values of the expression.

# Python Strings

- String is a **collection of characters**
- String literals in python are surrounded by either **single quotation marks**, or **double quotation marks**.
- 'hello' is the same as "hello".
- ~~• Strings can be output to screen using the print function. For example: print("hello").~~

# Python Strings

- ~~Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.~~

However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

# Python Strings

- **Multiline strings:**

- You can assign a multiline string to a variable by using three quotes
- Strings are arrays
- ~~Looping~~
- Len function
- Indexing
- `a = "Hello, World"`

# Python Strings

```
first_python_program.py x my_first_program.py x
s1 = "Hi 'Anil' How are you doing?"
s2 = 'Hello "Sachin" How are you doing?'
s3 = "Hello \"Dravid\" How is your day?"
# Useful to avoid the PEP8 error when your string is >128 characters can split with \
s5 = "Hello Country How is your day so go kl dbu 7 seen man kodi a man go " \
     "where do you go? then go to where are you goes into well man"
```

```
my_first_program x
"D:\Training\Python\Python Programs\venv\Scripts\python.exe" "D:/Training/Python/Python Prog
Hi 'Anil' How are you doing?
Hello "Sachin" How are you doing?
Hello "Dravid" How is your day?
Hello Country How is your day so go kl dbu 7 seen man kodi a man go where do you go? then go
Process finished with exit code 0
```

# Python Collections

- If you want to store more than 1 value, then we go for Collections.
- Integer, float can store one value , if you want to store more than 1 value should use collection. Collection of the different data types.

# Python Collections

- There are **four** collection data types in the Python programming language:
- **List** is a collection which is **ordered** and **changeable** (**mutable**). Allows duplicate members.
- **Tuple** is a collection which is **ordered** and **unchangeable** (**immutable**). Allows duplicate members.
- **Set** is a collection which is **unordered** and **unindexed**. **No duplicate members**.
- **Dictionary** is a collection which is **unordered**, **changeable** and **unindexed**. **No duplicate members**.

# Python Collections (List)

- A list is a collection which is **ordered** (consecutive memory blocks) and **changeable** (mutable). In Python lists are written with **square brackets**.
- It accepts **homogeneous** and **heterogeneous** data
- It allows **duplicate values**
- Show the 2 types of syntax to create empty list and list of values
- Indexing is possible
- To store more than one value, then we go for list. Represented with [] and each element is separated by ,



# Python Collections (**List**)

- **Access list Items**
- You can access the list items by referring to the index number, inside square brackets
- Show how to change the value
- Show the len()
- Show memory diagram
- Cover the index out of range error

# Python Collections (List)

- **Nested list**

# Python Collections (List)

- **Nested list**

# Python Collections (Tuple)

- A tuple is a collection which is ordered and **unchangeable (Immutable)**. In Python tuples are written with **round brackets**.
- It accepts homogeneous and heterogeneous data
- It allows duplicate values
- Show the 2 types of syntax to create empty tuple and tuple of values
- How to create a tuple with single element
- Indexing is possible

# Python Collections (Tuple)

## Access tuple Items

- You can access the tuple items by referring to the index number, inside square brackets
- Show how we can't change the value
- Show memory diagram
- **Notes:**
  - Creation of empty tuple is no where used
  - In case variable has to be shown which of immutable type then we can give an example of empty tuple variable

# Python Collections (Set)

- A set is a collection which is unordered (**non-consecutive memory blocks**) and unindexed, **No duplicate members.** In Python sets are written with **curly brackets.**
- It accepts homogeneous and heterogeneous data
- **It doesn't allow duplicate values**
- **Set is mutable,** but set values should be immutable type
- You can't change the existing values in set, but you can add the new values
- Show the 1 type and second type doesn't support of syntax to create empty set and set of values
- the elements contained in the set must be of an immutable (hashable) type.
- Indexing is not possible

# Python Collections (Set)

- In Python, any immutable object (such as an integer, boolean, string, tuple) is hashable, meaning its value does not change during its lifetime. This allows Python to create a unique hash value to identify it, which can be used by dictionaries to track unique keys and sets to track unique values.

# Python Collections (Set)

- **Access Items**

- You cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop or ask if a specified value is present in a set, by using the **in** keyword.
- **Note:** Sets are unordered, so the items will appear in a random order. You can't guess the order of set. So indexing is not possible
- Show memory diagram



# Python Collections (Set)

- **Frozen sets** in Python are **immutable objects** that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. **While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.**

If no parameters are passed, it returns an empty frozenset.

# Python Collections (Set)

- # working of a FrozenSet
- # Creating a Set
- String = ('G', 'e', 'e', 'k', 's', 'F', 'o', 'r')
- Fset1 = frozenset(String)
- print("The FrozenSet is: ")
- print(Fset1)
- 
- # To print Empty Frozen Set
- # No parameter is passed
- print("\nEmpty FrozenSet: ")
- print(frozenset())

# Python Collections (Set)

- **Very important :**
- On sets we can use the following operations instead of intersection, union and difference
- ```
>>> s={1,2,3}
```
- ```
>>> s2={3,4,5}
```
- ```
>>> s&s2
```

```
{3}
```
- ```
>>> s|s2
```

```
{1, 2, 3, 4, 5}
```
- ```
>>> s-s2
```

```
{1, 2}
```

# Python Collections (Dictionary)

- A dictionary is a collection which is **unordered**, **changeable** and **indexed by keys**. In Python dictionaries are written with **curly brackets**, and they **have keys and values**. **Indexing is not possible**.
- **Keys of a Dictionary must be unique** and of **immutable data type** such as Strings, Integers and tuples, but the **key-values can be repeated** and be of any type.
- **Nested dictionary**

# Python Collections (Dictionary)

- **Accessing Items**

- You can access the items of a dictionary by referring to its key name, inside square brackets:
- **Note:** There is also a method called `get()` that will give you the same result:
- Show the 2 types of syntax to create empty dictionary and dictionary of values
- Dictionary doesn't store the duplicate values instead it updates with last value
- Show memory diagram
- **Note:** From python version 3.7 dictionary order is guaranteed to be insertion order.
- **Important Note:** all arrays are iterable

# Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- **Single line comments**
- You can use '#' to create a single line comment
- A line of code starts with '#' will be ignored by Python. They are meant for fellow programmers
- Shortcut to more than one line to comment and uncomment
- Ctrl+/
  - Ctrl+Shift+/\*
  - Ctrl+Shift+\*/

# Python Comments

- **Multiline Comments**

- We can use `#` at the beginning of each line of comment on multiple lines
- Here, each line is treated as a single comment, and all of them are ignored.
- Another way of doing this is to use triple quotes, either `'''` or `"""`.

# Python Comments

- These triple quotes are generally used for multi-line strings. But if we do not assign it to any variable or function, we can use it as a comment.
- The interpreter ignores the string that is not assigned to any variable or function.



# Python Comments

- **Use of Python Comments**

1. Make Code Easier to Understand

# Python Collections (**List** vs **Tuple**)

| List                                              | Tuple                                                                 |
|---------------------------------------------------|-----------------------------------------------------------------------|
| Mutable                                           | Immutable                                                             |
| Slower                                            | Faster                                                                |
| Occupies more memory                              | Less memory                                                           |
| Memory is dynamic and it is not known to the user | Memory is fixed and it is known to user how much memory it can occupy |
| Unexpected changes can occur                      | Unexpected changes can't occur due to immutability                    |

When you are working with big data sets it can make the difference

# Difference between Collection datatypes

| List                                                                     | Tuple                                                                     | Set/frozen set                                                               | Dictionary                                                                                               |
|--------------------------------------------------------------------------|---------------------------------------------------------------------------|------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| List will be created with the <code>[]</code>                            | Tuple will be created with the <code>()</code>                            | Set will be created with the <code>{}</code><br>If the <code>{2,'hi'}</code> | Dictionary will be created with the <code>{}</code><br>If it contains <code>{'age':2,'text':'hi'}</code> |
| Ordered and indexed                                                      | Ordered and indexed                                                       | Unordered and unindexed                                                      | Unordered and unindexed<br><code>D['key name']</code>                                                    |
| In memory list values are stored in a <b>consecutive</b> memory location | In memory tuple values are stored in a <b>consecutive</b> memory location | In memory set values are stored in a <b>non-consecutive</b> memory location  | In memory dictionary values are stored in a <b>non-consecutive</b> memory location                       |

# Difference between Collection datatypes

| List                                     | Tuple                                     | Set/frozen set                              | Dictionary                                                              |
|------------------------------------------|-------------------------------------------|---------------------------------------------|-------------------------------------------------------------------------|
| Mutable                                  | Immutable                                 | Mutable/Immutable                           | Mutable                                                                 |
| List values can be mutable and immutable | Tuple values can be mutable and immutable | Set values should be of immutable data type | Dictionary keys should be of immutable data type and values can be both |
| Allows the duplicate values              | Allows the duplicate values               | It doesn't allow the duplicate values       | It doesn't allow the duplicate keys and It allows duplicate values      |

# Difference between Collection datatypes

| List                                                              | Tuple                                                             | Set/frozen set                                                                 | Dictionary                                                                     |
|-------------------------------------------------------------------|-------------------------------------------------------------------|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Memory is not efficiently used<br>It needs memory in the sequence | Memory is not efficiently used<br>It needs memory in the sequence | Memory is efficiently used<br>It doesn't require in the memory in the sequence | Memory is efficiently used<br>It doesn't require in the memory in the sequence |
| Faster for accessing the elements                                 | Faster for accessing the elements                                 | We can't access the elements in set, just we check value is present or not     | Comparatively slow in accessing the elements                                   |
| Difficult to read and remember                                    | Difficult to read and remember                                    | Difficult to read and remember                                                 | Easy to read and remember                                                      |

# Choosing a right collection data type is important

- When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data could increase the efficiency or security.
- **Realtime Examples:**
  - You can use the **List** for storing **persons age**, as age **changes** every year.
  - You can use the **Tuple** for storing **persons DOB**, as DOB **can't change** in a lifetime
  - You can use the **sets** for storing **Aadhar numbers**, as Aadhar number is Unique (**no duplicate Aadhar numbers allowed**)
  - If you want to have a **meaningful name (key)** to a value, we can use the **Dictionary**. Dictionary keys helps to **easily ready and remember** the values

# Difference between different types of Collections

In list/tuple we need to remember the value type and need to remember the index. Whereas dictionary has key associated with the value. It's easy to remember the key rather index

# Type casting

- **Specify a Variable Type**
- There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such **it uses classes to define data types, including its primitive types.**
- **Casting in python is therefore done using constructor functions:**
- `int()` - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals
- `Bool()`
- `Complex()`



# Type casting

```
>>> s1=' '  
>>> bool(s1)  
False  
>>> s1='Anil'  
>>> bool(s1)  
True  
>>> s2=None  
>>> bool(s2)  
False  
>>> bool(2)  
True  
>>> bool(100)  
True  
>>> bool(-1)  
True  
>>> bool(0)  
False
```

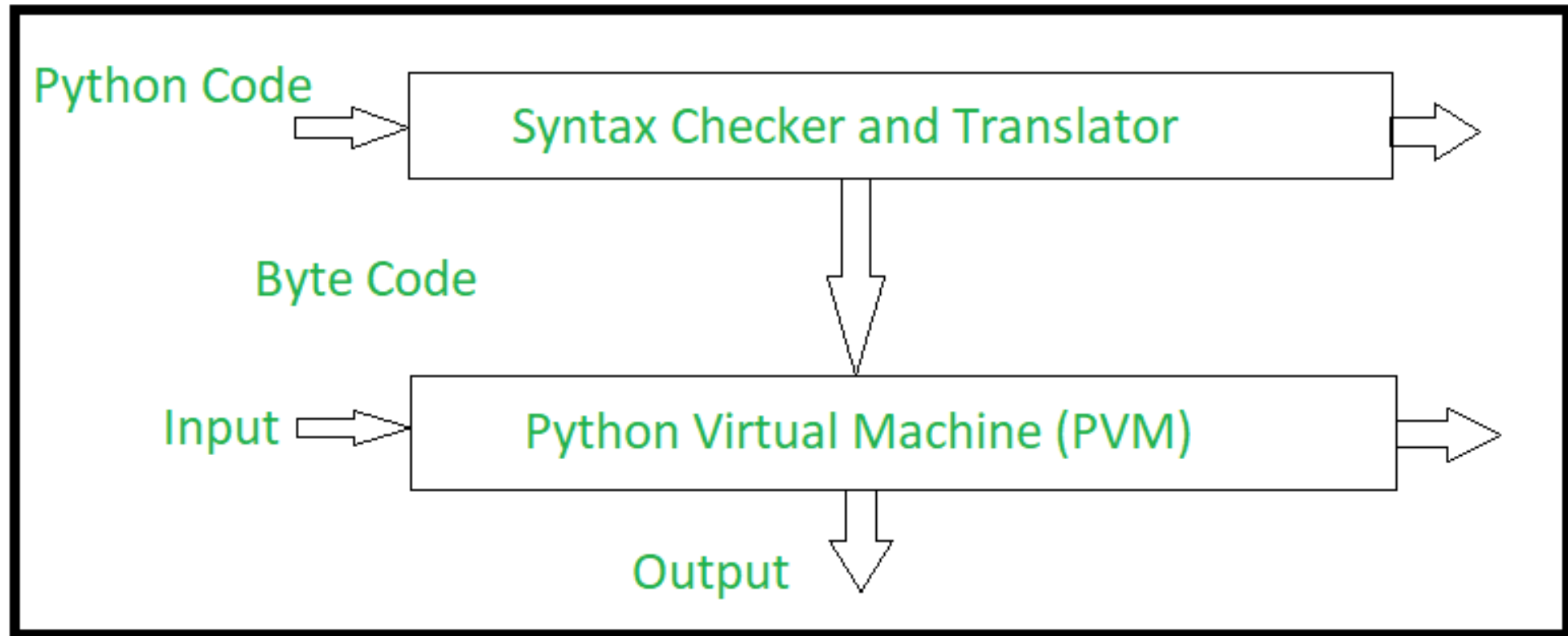
# Type casting

```
>>> s1=' '  
>>> bool(s1)  
False  
>>> s1='Anil'  
>>> bool(s1)  
True  
>>> s2=None  
>>> bool(s2)  
False  
>>> bool(2)  
True  
>>> bool(100)  
True  
>>> bool(-1)  
True  
>>> bool(0)  
False
```

# Literals, Hashable

- Literals in Python is defined as the raw data assigned to variables or constants while programming. We mainly have five types of literals which includes string literals, numeric literals, boolean literals, literal collections and a special literal None.
- Hashable=Immutable
- Unhashable=mutable

# Program Execution



# Slicing

- **Dictionary meaning of slice:**
- a flat piece of food that is cut from a larger piece
- a part of something
- In Python we are going to extract the part or piece from a iterables like string/list/tuple
- Slicing can be done with the data types where indexing is supported
- So, set and dictionary doesn't support the indexing. So, slicing can't be done with the set and dictionary

# Slicing

- **# Syntax**

- Variable\_name[start\_index:stop\_index:step\_index]
- **start\_index**: An integer number specifying at which position to start the slicing. Default is 0. **Start\_index is inclusive**
- **stop\_index**: An integer number specifying at which position to stop the slicing. Default is end of the collection. **Stop\_index is exclusive**
- **Step**: An integer number specifying the step of the slicing. Default is 1
- Alternatively, you can use the slice() function also

## Slicing (Important points)

- Python by default look from left to right (i.e., step index 1)
- If you want to look from right to left, then we need to specify (step index as -1)

Variable\_name[start\_index:stop\_index:step\_index]

If the step value is +ve then stopindex+1

If the step value is -ve then stopindex-1

If the both start and stop index is not provided, then it returns all the data items in the iterable

# Slicing and Indexing (Real time examples)

- Real time examples:
- Your data as follows
- ['Anilkumar','Para','Sachin','Tendulkar','Dravid','Rahul']
- I am getting the in above format i.e. First name and last name and I want to extract all the last names we can use the slicing indexing [::2]
- Let's say in source data we always want the latest data
- [2,3,4,5,6]
- [-1]



# Operators

- Operators are special symbols which performs operations on operands.
- Operands can be a variable or value.

# Operators

- Python divides the operators in the following groups:
  - Arithmetic
  - Relational/comparison
  - Assignment
  - Logical
  - Membership (in)
  - Identity (is)
  - Bitwise

# Arithmetic

- + : **addition** or **concatenation** :
  - When you perform with Numeric and Boolean types (numbers, it does mathematical calculation)
  - When you perform with same grouped data type it does concatenation (except set, dictionary).
  - list + tuple concatenation not possible.
  - Only same type + works.
- - : Subtraction: only numeric, Bol and set data types

# Arithmetic

- \*: multiplication/replication: When you perform with Numeric type it does multiplication. Replication is same set of values repeating again and again. Replication is possible with integer type with string, list, tuple, (except set and dictionary) and not with string with collections and vice versa.
- Replication is not possible with set why? Set doesn't allow duplicate values

# Arithmetic

- `/`: True division: always returns decimal values in the result/output
- `//`: Floor division: always returns integer values in the result/output if both the dividend divisor is integers and i.e., `floor(value)`
- `%`: modulus: returns the remainder
- `**` : Exponentiation It is used to raise the first operand to power of second. Can use `pow()`

# Relational Operators

- Relational Operators are the **operators used to create a relationship and compare the values of two operands**. Result would be True or False
- For example, there are two numbers, 5 and 15, and if you want to check  $5 > 15$  or not? Then you can use the relational operators

# Relational Operators

| Operator | Description              |
|----------|--------------------------|
| ==       | Equal                    |
| !=       | Not equal                |
| >        | Greater than             |
| <        | Less than                |
| >=       | Greater than or equal to |
| <=       | Less than or equal to    |

# Relational Operators

- $5+4j>2$
- `TypeError: '>' not supported between instances of 'complex' and 'int'`
- $2>'Hi'$
- `TypeError: '>' not supported between instances of 'int' and 'str'`
- $5+4j>4+5j$
- `TypeError: '>' not supported between instances of 'complex' and 'complex'`



# Relational Operators

- `'Hi'>'Hi'` => It works
- `'Hi'>['Hi']`
- `TypeError: '>' not supported between instances of 'str' and 'list'`
- `[1,2,3]>[1,2]` => It works
- `(1,2,3)>(1,2)` => It works
- `{1,2}>{1,2,3}` => It works
- `{'name':'Anil'}>{'name':'Anil','Age':33}`
- `TypeError: '>' not supported between instances of 'dict' and 'dict'`

# Assignment Operators

• = += -= \*= /= %= //= \*\*= &= |= ^= >>= <<=

# Logical Operators

- A **logical operator** is a word used to connect two or more expressions
- Logical operators combine the conditional statements
  - and
  - or
  - not
  - nand (not and) =  $\text{not}(\text{op1 and op2})$
  - nor (not or) =  $\text{not}(\text{op1 or op2})$

# Truth Table

| Input |   | Output  |        |       |          |         |
|-------|---|---------|--------|-------|----------|---------|
| A     | B | A and B | A or B | not A | A nand B | A nor B |
| 0     | 0 | 0       | 0      | 1     | 1        | 1       |
| 0     | 1 | 0       | 1      | 1     | 1        | 0       |
| 1     | 0 | 0       | 1      | 0     | 1        | 0       |
| 1     | 1 | 1       | 1      | 0     | 0        | 0       |

**Note:** If the operands are numeric values, then python returns numeric values, otherwise it returns Boolean values .

# Membership Operators

- In & not in
- Membership operators are used to check whether the value or data is present in group or not. Returns true if it is present else false
- **Syntax:**
- DATA in group
- **Note:** In Dictionary it check only keys and not the values

# Identity Operators

- **is** & **is not**
- Identity operator is used to check whether two variables are point to same memory location or not
- If you want to check the address of variable syntax is  
    `id(var)`
- Is identifier being a related function i.e., used to check whether the content present inside the string can be a name given to memory location or not?

# Identity Operators

- Difference between `copy` and `=` assignment operator
- `Collection2 = collection1.copy()`
- `Collection2 = collection1`

# Difference between == and is operator

| ==                                                          | is                                                                         |
|-------------------------------------------------------------|----------------------------------------------------------------------------|
| Checks the <b>value</b> of the variables for the comparison | Checks the <b>memory location (address)</b> of variable for the comparison |

Every time you define an object in Python, you'll create a new object with a new identity. However, for the sake of optimization (mostly) there are some exceptions for small integers (between -5 and 256) and interned strings which are singletons and have the same id (one object with multiple pointers).



# String Interning in Python

The string interning in Python is a mechanism of storing only one copy of a string value in the memory. If there are a few string variables whose values are the same, they will be interned by Python implicitly and refer to the same object in the memory.

This mechanism introduces two advantages:

1. Save the space in the memory

2. Save the time when comparing strings whose values are the same

The first one is obvious, since storing only one copy needs less space than storing all of them.

The second one is because if two strings refer to the same object, they are definitely equal to each other and there is no need to compare the characters of them one by one.

# String Interning in Python

For example, up until the version of Python 3.7, the peephole optimization is used for string interning and all strings longer than 20 characters will not be interned. However, the algorithm was changed to the AST optimizer then, and the length is equal to 4096 rather than 20.

## Conclusion

The string interning mechanism is a hidden gem in Python. We may don't feel it for the first time, since Python will intern some strings implicitly. But if we totally understand it and use it explicitly and properly, it will be of great value to improve the speed of string comparisons.

# Interning in Python (Contd.)

- In computer science, **interning** is re-using objects of equal value on-demand instead of creating new objects. This **creational pattern** is frequently used for **numbers and strings** in different programming languages.
- In many object-oriented languages such as **Python**, even **primitive types** such as **integer numbers are objects**. To avoid the overhead of constructing a large number of integer objects, these objects get reused through interning.

# Interning in Python

- For interning to work **the interned objects must be immutable**, since state is shared between multiple variables. String interning is a common application of interning, where many strings with identical values are needed in the same program.

# is operator outputs for Immutable data types

| Type/Range                        | IDLE  | PyCharm/terminal |
|-----------------------------------|-------|------------------|
| int (-5 to 256)                   | True  | True             |
| int (number < -5 or number > 256) | False | True             |
| float                             | False | True             |
| complex                           | False | True             |
| bool                              | True  | True             |
| String (<=4096 characters)        | True  | True             |
| String (>4096 characters)         | False | False            |
| tuple                             | False | True             |

## is operator outputs for Mutable data types

| Type/Range | IDLE  | PyCharm/terminal |
|------------|-------|------------------|
| list       | False | False            |
| set        | False | False            |
| dictionary | False | False            |

# Number System

- Number Systems — Decimal, Binary, Octal and Hexadecimal
- Base 10 (*Decimal*) — Represent any number using 10 digits [0–9]
- Base 2 (*Binary*) — Represent any number using 2 digits [0–1]
- Base 8 (*Octal*) — Represent any number using 8 digits [0–7]
- Base 16(*Hexadecimal*) — Represent any number using 10 digits and 6 characters [0–9, A, B, C, D, E, F]

# Bitwise

- Bitwise operators are used to perform operations on binary representation of values



# Bitwise

| Operator | Name                                                         | Description                                                                                             |
|----------|--------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| &        | Bitwise AND                                                  | Sets each bit to 1 if both bits are 1                                                                   |
|          | Bitwise OR                                                   | Sets each bit to 1 if one of two bits are 1                                                             |
| ^        | Bitwise XOR                                                  | Sets each bit to 1 if <b>only</b> one of two bits are 1                                                 |
| ~        | Bitwise NOT or Complement<br><a href="#">unary operation</a> | <b>Invert the sign</b> , <b>number+1 for positive value</b> and <b>number- 1 for negative value</b>     |
| <<       | Bitwise left shift<br>(Zero fill)                            | Shift left by pushing copies of the rightmost bit in from the right, and let the leftmost bits fall off |
| >>       | Bitwise right shift<br>(Signed)                              | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off |

XOR: the resulting bit evaluates to one if exactly *one* of the bits is set.

# Bitwise

- In right shift operation for every shift number will be floor division i.e.  $(\text{number}) // (2^n)$  (number of shifts)
- In left shift operation results equal to given number  $\times 2^{\text{power } n}$  (number of shifts)
- How bitwise  $\sim$  works is if it is positive sign then output is negative sign number+1
- How bitwise  $\sim$  works is if it is negative sign then output is positive sign number-1
- **Note** : Shifting is only performed with numbers

| Operator                                                                                           | Description                                                                        |
|----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <code>(expressions...),</code><br><code>[expressions...], {key: value...}, {expressions...}</code> | Binding or parenthesized expression, list display, dictionary display, set display |
| <code>x[index], x[index:index], x(arguments...),</code><br><code>x.attribute</code>                | Subscription, slicing, call, attribute reference                                   |
| <code>**</code>                                                                                    | Exponentiation                                                                     |
| <code>+x, -x, ~x</code>                                                                            | Positive, negative, bitwise NOT                                                    |
| <code>*, @, /, //, %</code>                                                                        | Multiplication, matrix multiplication, division, floor division, remainder         |
| <code>+, -</code>                                                                                  | Addition and subtraction                                                           |
| <code>&lt;&lt;, &gt;&gt;</code>                                                                    | Shifts                                                                             |
| <code>&amp;</code>                                                                                 | Bitwise AND                                                                        |
| <code>^</code>                                                                                     | Bitwise XOR                                                                        |
| <code> </code>                                                                                     | Bitwise OR                                                                         |
| <code>in, not in, is, is not, &lt;, &lt;=, &gt;, &gt;=, !=, ==</code>                              | Comparisons, including membership tests and identity tests                         |
| <code>not x</code>                                                                                 | Boolean NOT                                                                        |
| <code>and</code>                                                                                   | Boolean AND                                                                        |
| <code>or</code>                                                                                    | Boolean OR                                                                         |
| <code>=</code>                                                                                     | Assignment expression                                                              |

# Operator Precedence (order, importance/Priority)

| Operator                                                                       | Description                                                                        |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| (expressions...),<br>[expressions...],<br>{key: value...},<br>{expressions...} | Binding or parenthesized expression, list display, dictionary display, set display |
| x[index], x[index:index],<br>x(arguments...), x.attribute                      | Subscription, slicing, call, attribute reference                                   |
| **                                                                             | Exponentiation (raise to the power)                                                |
| ~, +, -                                                                        | Complement, unary plus and minus<br>(Example: ~2, +2, -2)                          |

# Operator Precedence (order, importance/Priority)

| Operator                                                              | Description                                         |
|-----------------------------------------------------------------------|-----------------------------------------------------|
| <code>*, /, %, //</code>                                              | Multiplication, division, floor division, remainder |
| <code>+, -</code>                                                     | Addition and subtraction                            |
| <code>&lt;&lt;, &gt;&gt;</code>                                       | Bitwise Right shift and bitwise left shift          |
| <code>&amp;</code>                                                    | Bitwise 'AND'                                       |
| <code>^</code>                                                        | Bitwise 'XOR'                                       |
| <code> </code>                                                        | Bitwise 'OR'                                        |
| <code>in, not in, is, is not, &lt;, &lt;=, &gt;, &gt;=, !=, ==</code> | Membership, identity, Comparisons                   |

# Operator Precedence (order, importance/Priority)

| Operator                               | Description         |
|----------------------------------------|---------------------|
| not                                    | Logical/Boolean not |
| and                                    | Logical/Boolean and |
| or                                     | Logical/Boolean or  |
| =, +=, -=, *=, /=, %=, //=, **=<br>**= | Assignment          |

# Operator Precedence (order, importance/Priority)

- ‘\*\*’ operator has **right to left Associativity** and other operators has **left to right Associativity**

# Python Indentation

- Most of the programming languages like C, C++, Java use braces {} to define a block of code. Python uses indentation.
- A code block (body of a function, loop etc.) starts with indentation and ends with the first un-indented line. The amount of indentation is up to you, but it must be consistent throughout that block.
- Generally, four whitespaces are used for indentation and is preferred over tabs.
- What is the purpose of the colon before a block in Python?
  - The colon is there to declare the start of an indented block.



# Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.
- The number of spaces is up to you as a programmer, but it has to be at least one.
- You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

# Python Indentation

- Indentation is a very important concept of Python because without properly indenting the Python code, you will end up seeing **IndentationError** and the code will not get compiled.
- Python indentation refers to adding white space before a statement to a particular block of code. In another word, all the statements with the same space to the right, belong to the same code block

# Python Indentation

- All statements with the same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right. You can understand it better by looking at the following lines of code.

# Python Indentation

- Indentation is the leading whitespaces before any statement in python
- **Purposes**
- Improves readability
- Helps in indicating a block of code
- compare with the C {} and indentation is added and doesn't help in represents a block of code . helps in improving the readability
- Will use for writing any block of code like for, if, function, class, etc.

# Comments in Python

- # : single line comment
- ''' : multiline comments

# Control Statements

- Control Statements are **used to transfer the control from one part of the program to another depending on a condition.**
- Control statements control the flow of the execution.

# Control Statements

- We have **3 types** of **control statements**
  - Conditional (if, if else, nested if, nested if else, If elif ladder)
  - Loops (while, for)
  - Branching (break, continue)

# Conditional Control Statements

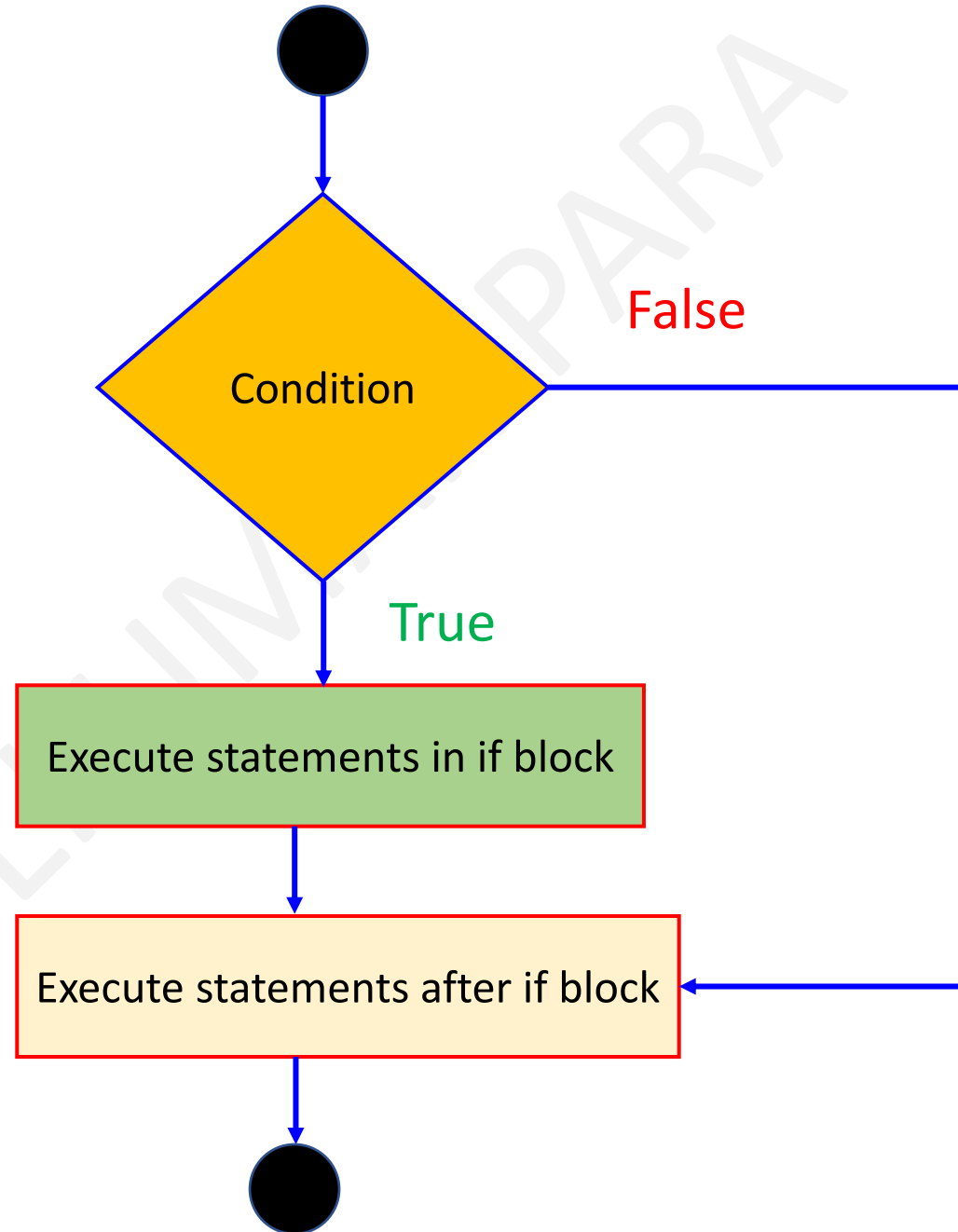
- If statement
- if else statement
- nested if
- Nested if else
- If **elif** ladder



# Conditional Statements Examples

- Let's say if you are driving and you reached traffic signal.
- If it is Green => Move
- If it is Red => Stop
- If it is Orange => prepare to stop

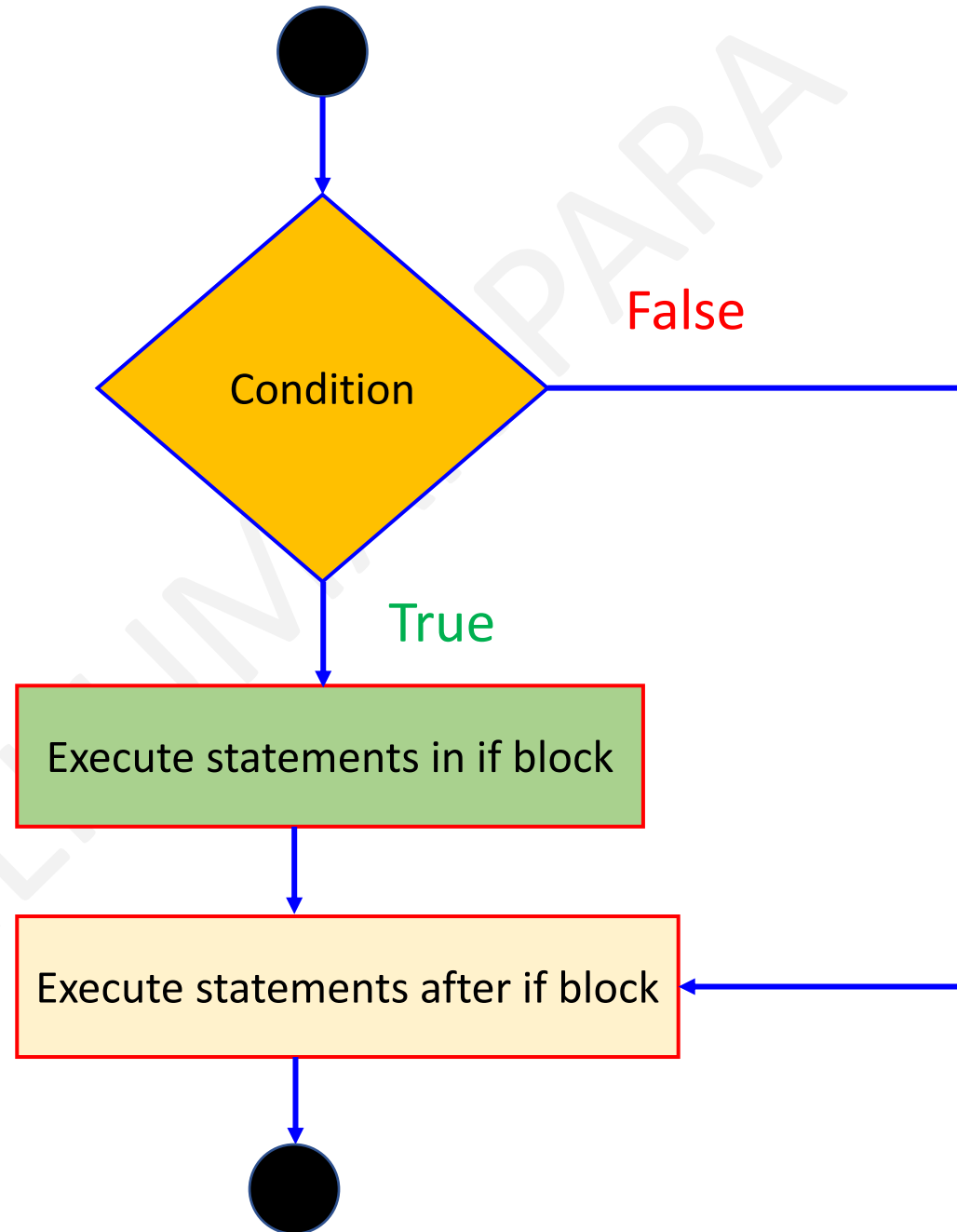
# If Statement



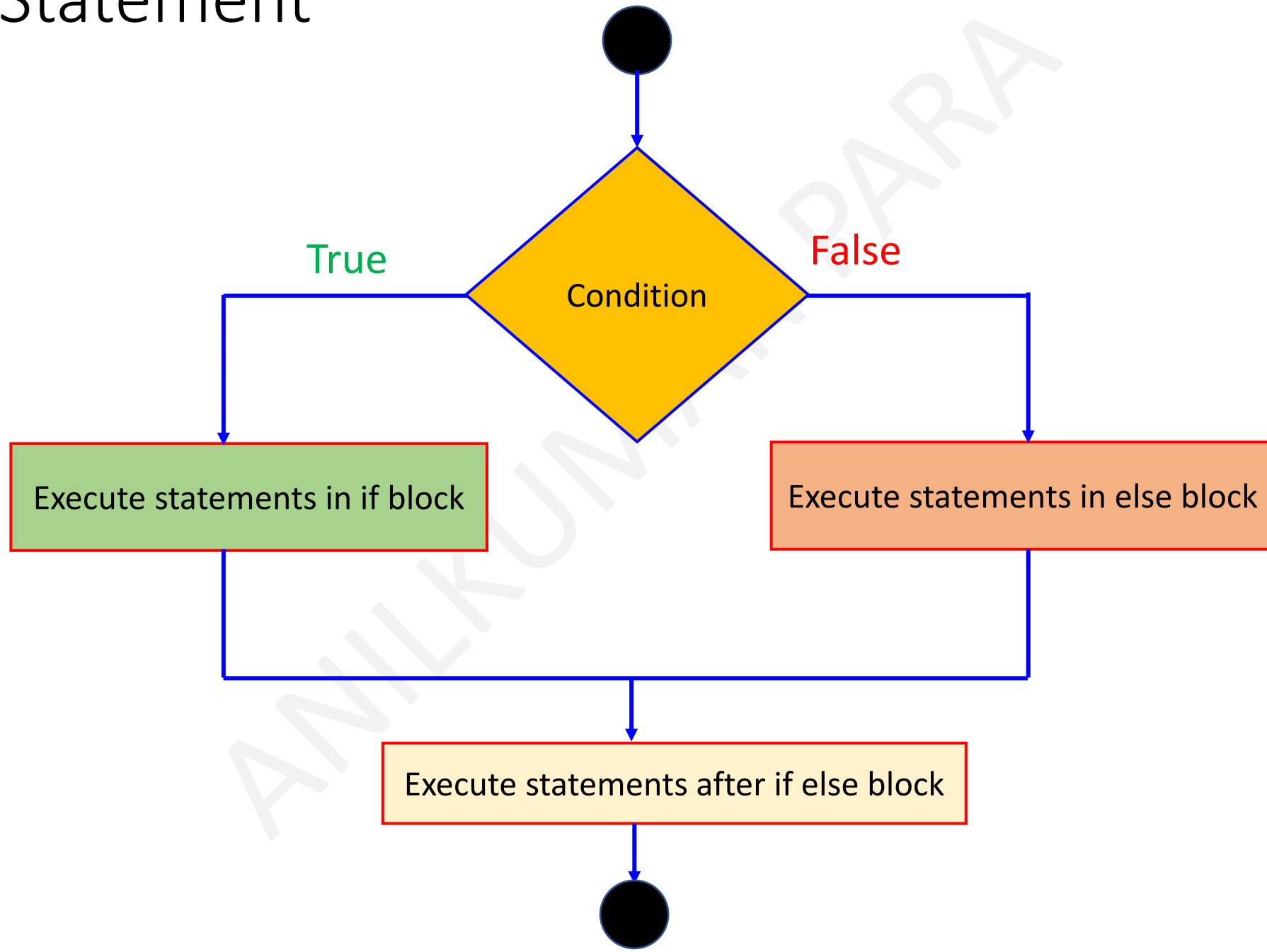
# If Statement

## Syntax:

- If condition:  
Block of code



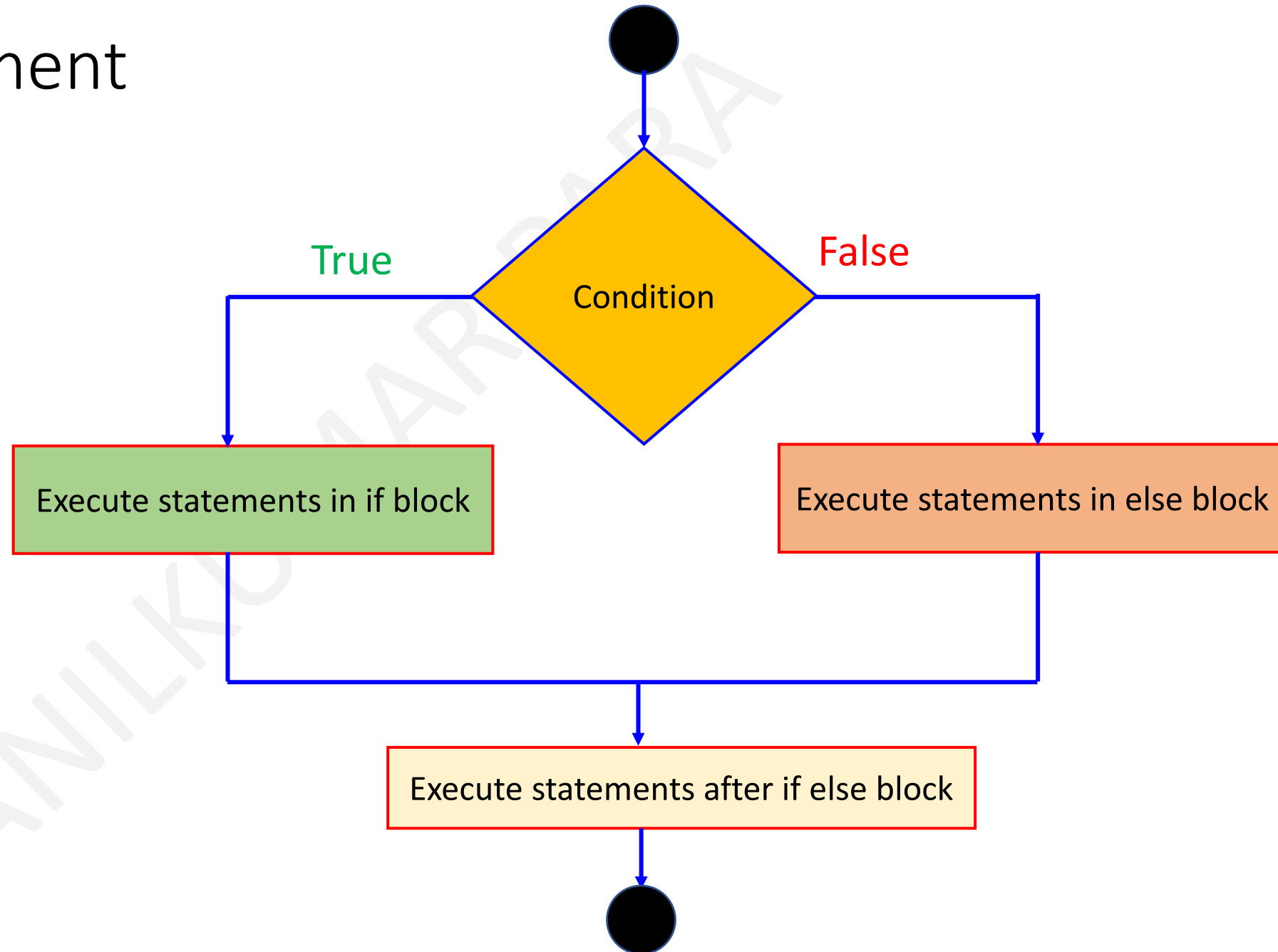
# If else Statement



# If else Statement

## Syntax:

- If condition:
  - Block of code
- Else:
  - Block of code



# Nested If Statement

- **Nested if** is **one if inside of another if**, allows you to test multiple criteria and increases the number of possible outcomes.

## **Syntax:**

- If condition:
  - If condition:
    - If condition:
      - Statement-1

## Nested If else

- **Nested if else** is **one if else inside of another if else**, allows you to test multiple criteria and increases the number of possible outcomes.

# Nested If else

## Syntax:

- If condition-1 :
  - If condition-2:
    - If condition-3:
      - Statement-1
    - Else:
      - Statement-2
  - Else:
    - If condition-4:
      - Statement-3
    - Else:
      - Statement-4



## If else ladder

- If else ladder is a conditional statement used for selection between multiple set of statements based on multiple test conditions. The various test conditions are provided inside each if statement

## If else ladder

- In else-if ladder the conditions are evaluated from the top of the ladder downwards. As soon as a true condition is found, the statement associated with it is executed skipping the rest of the ladder.

# If else ladder

- In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block

# If else ladder

- difference between the if-else and if-else-if ladder is, in the case of the if-else statement, you have only one expression, so based on that decision is taken. But in the case of the if-else-if ladder, you can evaluate multiple expressions.

# If else ladder

## Syntax:

- If condition:
  - Statement-1
- elif condition:
  - Statement-2
- elif condition:
  - Statement-3
- else:
  - Statement-4

# If else ladder

The **CBSE** Uses the following grading system.

| Letter Grade | Marks        |
|--------------|--------------|
| A1           | 90-100       |
| A2           | 80-90        |
| B1           | 70-80        |
| B2           | 60-70        |
| C1           | 50-60        |
| C2           | 40-50        |
| D            | 33-40        |
| E(Failed)    | 33 and below |

# Programs

- WAP to check whether person wants to go to movie or not?
  - If it is raining no movie, not raining , go to movie
- WAP to check whether a person can work as per government rules based on age of the person
- WAP to find a grade of student
- WAP to check whether person is in which country
- WAP to check person is in which area
- WAP to check person is studying BIPC [Biology, Physics, and Chemistry] or MPC [Maths Physic and Chemistry]

# Match case

- Unlike every other programming language, we have used before, Python does not have a switch or case statement.
- You can achieve using if else ladder
- One of the language's most recent additions is the **match-case statement**. Introduced in Python 3.10, it allows you to evaluate an expression against a list of values.



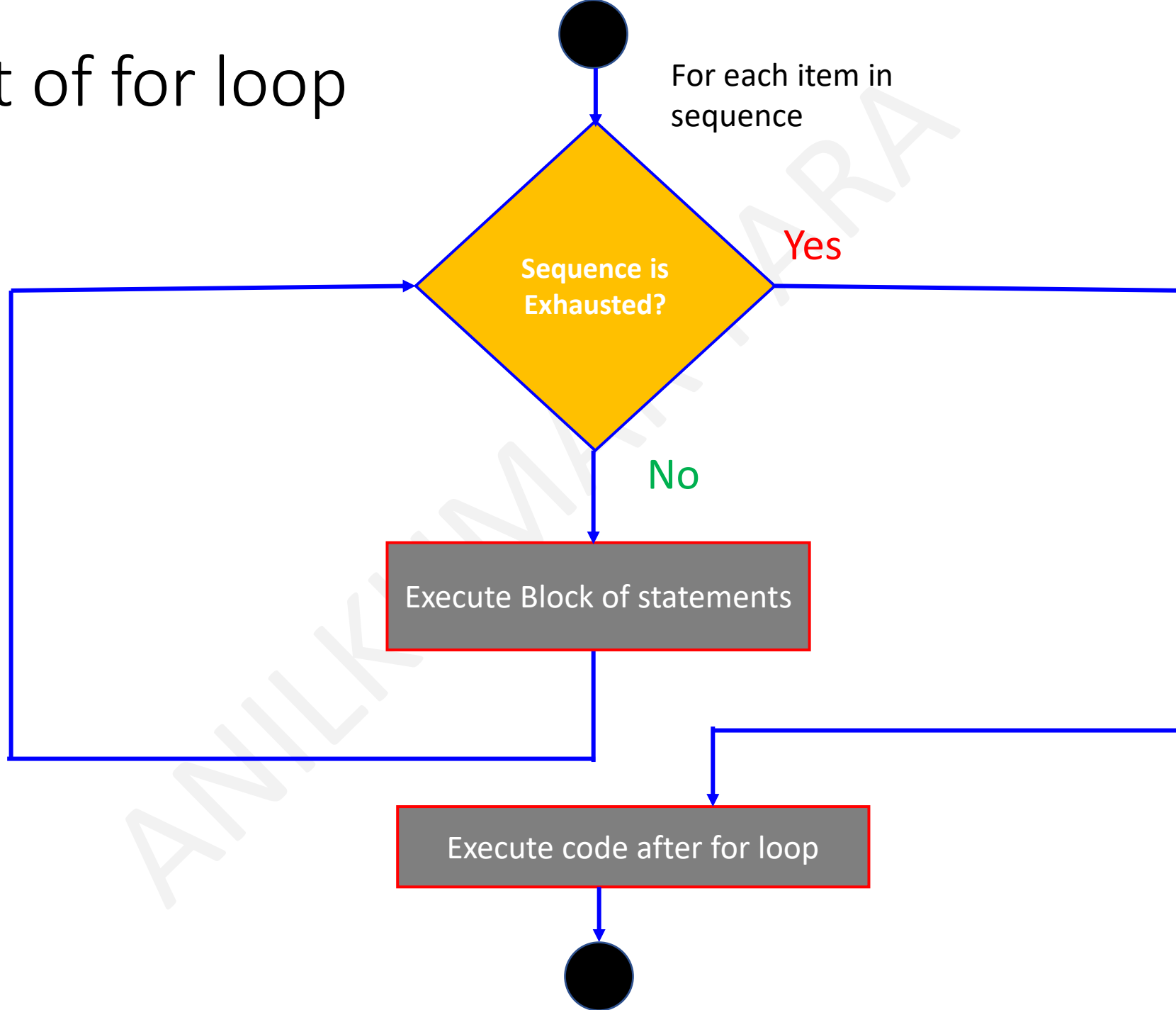
# Looping

- Looping means **repeating something over and over until a particular condition is satisfied.**
- **Realtime examples:**
  - Let's you want to listen to a song 5 times
  - Let's say you want to check the grade of every student in the class
  - Software program in a mobile device allows user to unlock the mobile with 5 password attempts. After that it will asks you to wait for 30 seconds, and you can try after 30 seconds

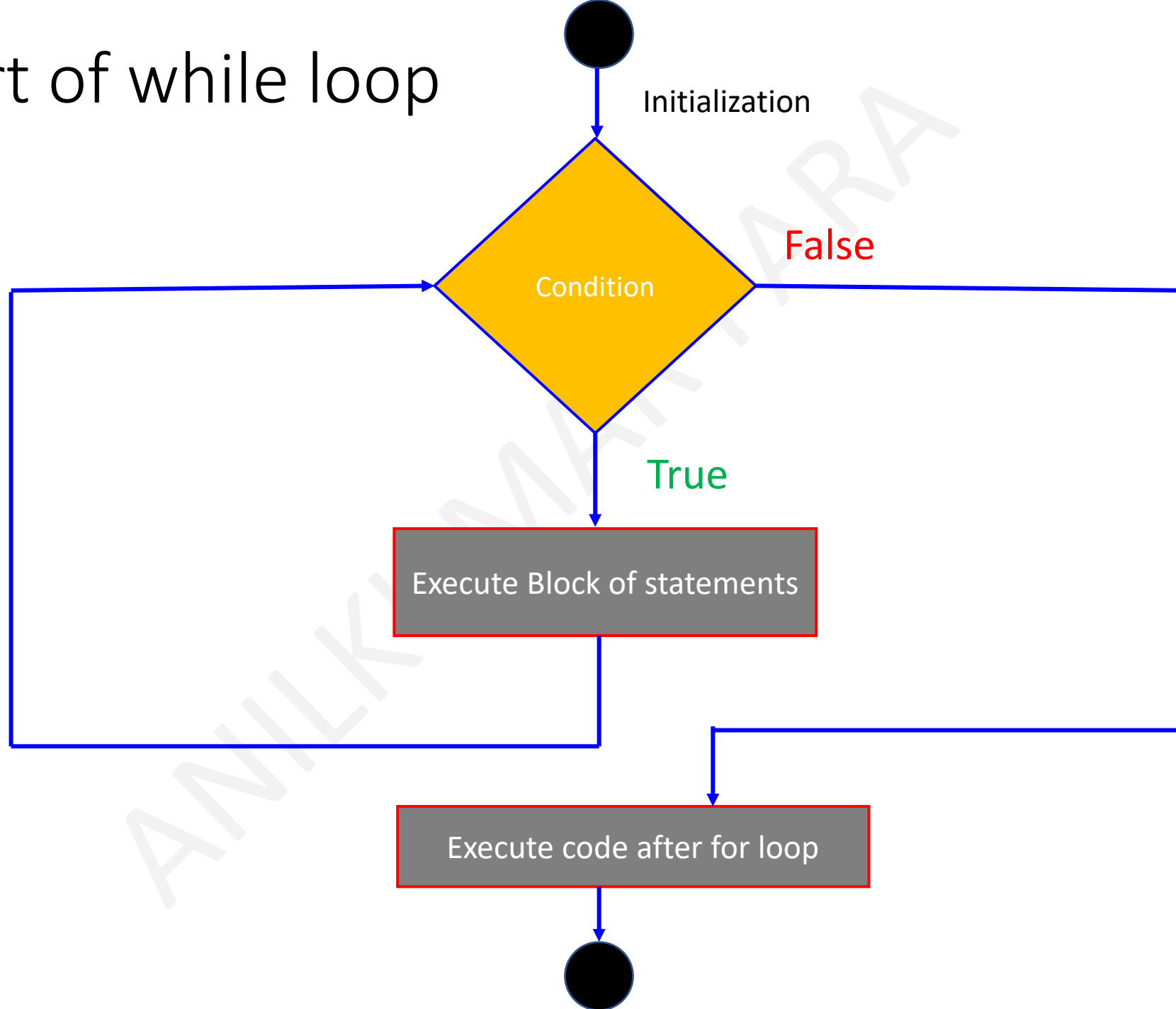
# Looping

- In Python we have 2 types of loops
  - for
  - while
- A for/while loop in Python is a control flow statement that is used to repeatedly execute a group of statements as long as the condition is satisfied. Such a type of statement is also known as an iterative statement.

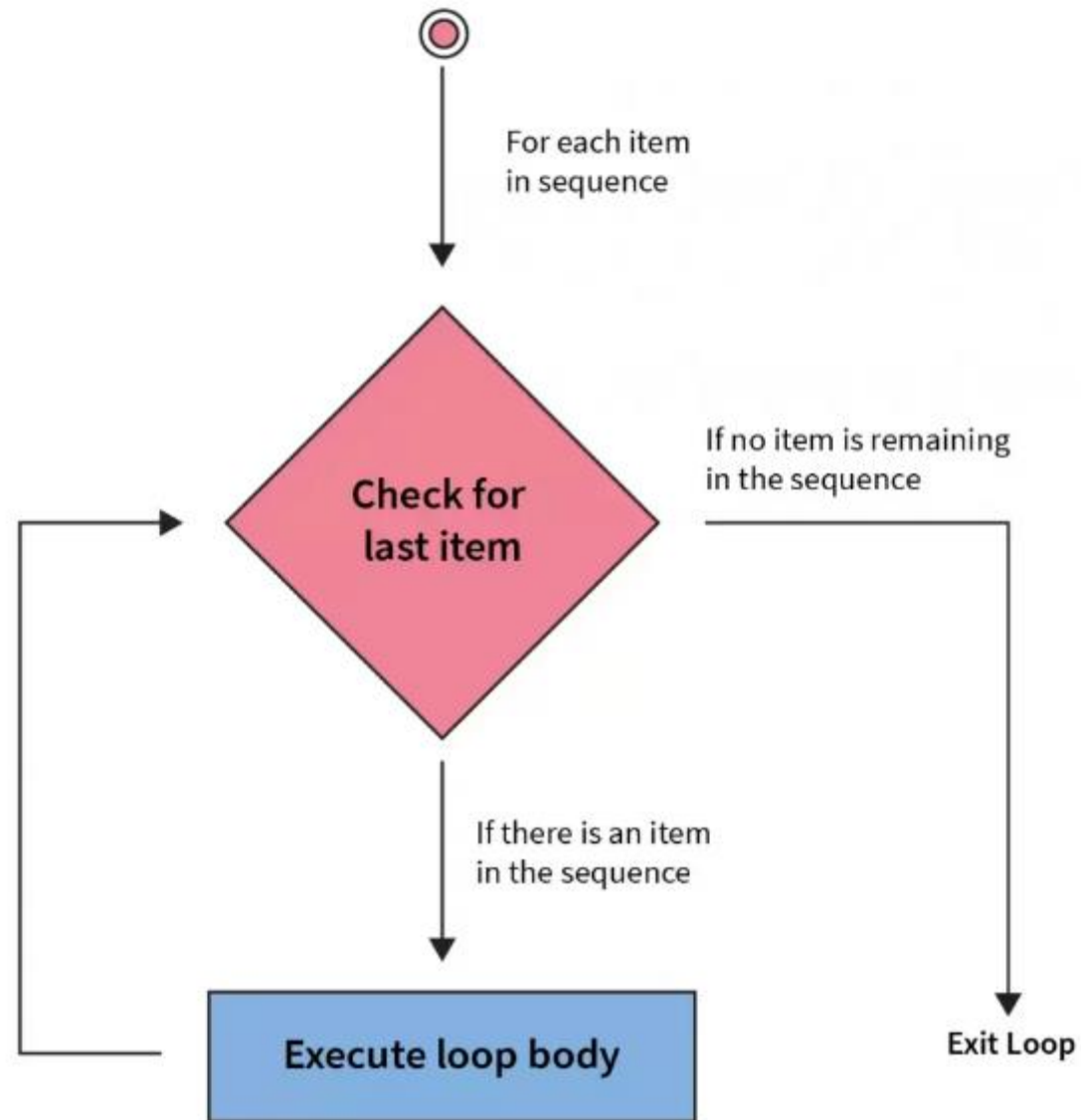
# Flowchart of for loop

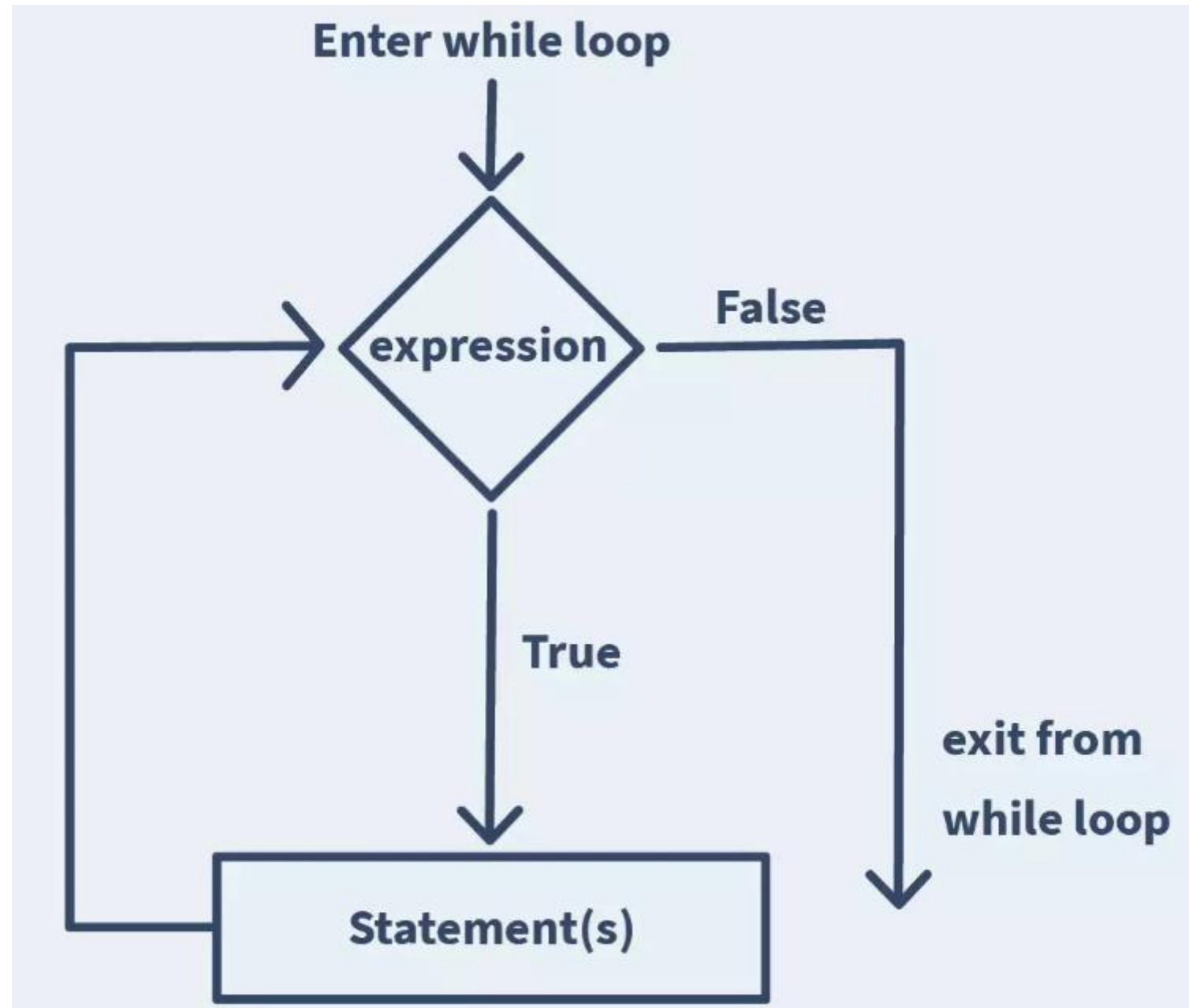


# Flowchart of while loop



# Flowchart of for loop





# Syntax of for and while

- **for** variable **in** group of values (collection/string):  
block of code
- Initialization
- **While** condition:  
block of code  
update value

# Dictionary values

```
>>> d={'name': 'A', 'RN': 20}
>>> d
{'name': 'A', 'RN': 20}
>>> for k in d:
...     print(k)
...     print(d[k])
...
...
name
A
RN
20
```



# Break, Continue

- 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

- **Break:**

- The **break** statement in Python **terminates the current loop** and resumes execution at the next statement, just like the traditional break found in C.
- Will see search example array search element

# Break, Continue

- **Continue:**

- The **continue** statement in Python returns the control to the beginning of the current loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- Will see simple print 1 to 4 and 8 in 1 to 8 numbers
- Go to supermarket and buy vegetables when they are fresh otherwise no

# Break, Continue

- Cover the else clause also with break
- Main use of else is used to avoid the flags, if element found or not you need to use the flag

# Break, Continue

```
>>> x=0
>>> while x<=10:
...     print(x)
...     x+=1
...     if x==8:
...         break
...     print("Executed all statements in the loop")
... else:
...     print("out of the while loop")
```

```
>>> x=0
>>> while x<=10:
...     print(x)
...     x+=1
...     print("Executed all statements in the loop")
... else:
...     print("out of the while loop")
```

# Range()

- **range()** is a function which generates and return a sequence of numbers in the specified range. It doesn't save in the memory. Hence it saves the memory. It generates numbers on the fly.
- In case if you want to create a list of numbers in sequence or even or odd. It's good to use the range function to save the memory instead of using the list, tuple

# Range()

- **Syntax:**
- `range(start, stop, step)`
- **start:** Optional. An integer number specifying at which number the sequence to start. Default is 0
- **stop:** An integer number specifying at which number the sequence to stop
- **Step:** Optional. An integer number specifying the step of the sequence. Default is 1

# for loop **vs** while

| Aspect                | For Loop                                                                                                             | While Loop                                                                                         |
|-----------------------|----------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <b>Data structure</b> | If you have a <b>ready-made data structure</b> (set, tuple, list, etc.) available, you can simply go for “for loop”. | If you <b>don't have a tidy data structure</b> to iterate through, then you must use ‘while loop’. |
| <b>Initialization</b> | Loop <b>variable is automatically initialized</b> based on the sequence.                                             | Loop variable <b>needs to be initialized</b> before the loop starts.                               |



# for loop **vs** while

| Aspect    | for Loop                                                           | while Loop                                                          |
|-----------|--------------------------------------------------------------------|---------------------------------------------------------------------|
| Condition | Condition is based on the elements in the sequence.                | Condition can be any expression that evaluates to True or False.    |
| Checking  | Condition is checked before each iteration.                        | Condition is checked before each iteration.                         |
| Execution | 'For loop' executes block of code until the sequence is exhausted. | 'While loop' executes a block of code until the condition is False. |

# for loop *vs* while

| Aspect         | for Loop                                                           | while Loop                                                               |
|----------------|--------------------------------------------------------------------|--------------------------------------------------------------------------|
| Update value   | You <b>don't</b> require to update the value of the loop variable. | You must <b>explicitly</b> handle the update value of the loop variable. |
| Termination    | Terminates when all elements in the sequence have been processed.  | Terminates when the condition becomes 'False'.                           |
| Infinite loops | No chance of going for the infinite loops                          | It can be prone to infinite loops if not handled correctly.              |

# for loop vs while

| Aspect      | for Loop                                                                            | while Loop                                                                       |
|-------------|-------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| Range()     | Range function can be used , but the relevant variable initialization is not needed | Range function can be used , but the relevant variable initialization is needed  |
| Readability | Generally, more readable and concise for iterating through known sequences.         | Less readable as it may require extra lines of code to manage the loop variable. |

# Nested loops

- a loop statement **inside** another loop statement.
- or
- loop **inside** loop.
- **a loop exists inside the body of another loop**, it's called a nested loop

# What is a function

- A function is simply a “chunk/block” of code that you can use repeatedly, rather than writing it out multiple times. Functions enable programmers to break down or decompose a problem into smaller chunks, each of which performs a particular task.
- Functions are written to perform specific task
- The only way without functions to reuse code is copying the code.
- Sometimes we may call functions as subroutines or procedures.

# Functions

- The main advantage of functions are reusability, readability due to Modularization
- There are 2 types of functions are available
  - **Built-in functions:** A function which is already defined in programming framework is called Built-in functions or predefined functions
  - **User defined functions:** Functions which are created by user as per his own requirements

# Functions

- **Syntax:**
- `def function_name(args):`  
    Body  
    Return

# Functions

- In program, functions are not executed until it is called
- The function definition must be provided before calling function
- Return statement and args are not mandatory
- Explain the calling and caller function



# Cover

- Function declaration
- Function definition
- Function call

# Different types of functions

- User defined functions are classified into 4 types
  1. Function **without** any **arguments** and **return type**
  2. Function **with** **arguments** and **without** **return type**
  3. Function **without** any **arguments** and **with** **return type**
  4. Function **with** any **arguments** and **return type**

# Different types of functions

- User defined functions are classified into 4 types
  1. Function with **No** Arguments and **No** Return Value
  2. Function with Arguments and **No** Return Value
  3. Function with **No** Arguments and a Return Value
  4. Function with Arguments and a Return Value

# Different types of functions

## 1. Function with **No** Arguments and **No** Return Value

1. This includes things like **printing to the console, writing to a file, modifying global or external variables, or triggering some other kind of action.**
- 2. Initialization:** A function might be called at the **start of a program** or system to set up certain conditions, initialize hardware, reset global variables, or prepare some resources.
3. They can be used for tasks that need to be performed repeatedly but don't necessarily require input parameters or return values like **logout of the account**

# Different types of functions

## 1. Function with Arguments and **No** Return Value

1. Modifying global or external state, **printing messages to the console, writing to a file, or updating a database entry.**
2. Such functions can be used to interact with the user. For example, a function might take a message as an argument and **display it in a specific format**
3. Data Manipulation: While they don't return values, these functions **can still modify the data they receive.** For instance, a function **might accept a list and sort it in place.** The list is changed, **but the function doesn't return a new list.**
4. you might have a function that takes **event details as arguments and then processes the event (e.g., handling a button click and receiving details about the button state).**

# Different types of functions

## 1. Function with **No** Arguments and Return Value

- 1. Resource Allocation:** A function might create and return a new database connection.
- 2. Time or Date:** Functions that return the current time or date are examples of this kind. They don't need any arguments to produce their result.
- 3. Generating Constants:** Such functions can be used to return constant values. For example, a function might always return a specific configuration setting or a default value.

# Different types of functions

## 1. Function with Arguments and Return Value

1. These are the most common types of functions.
2. **Data Processing:** These functions take **data as input, process it, and return a result**. For example, a function might take a string and return its length or transform a list based on certain criteria.
3. **Mathematical Computations:** Functions can be designed to perform mathematical operations, like calculating **factorial, power**, or any custom mathematical computation, and return the result.
4. **Decision Making:** They can be used to make decisions. For instance, **a function might accept an age value and return a boolean indicating if the age qualifies for a specific privilege**.

# Different types of functions

## 1. Function with Arguments and Return Value

- 1. Data Retrieval:** Such functions can be used to retrieve specific data from a data source. For instance, a function might accept a user ID and return the corresponding user's details from a database.
- 2. Validation and Transformation:** Functions can validate input data and transform it. For example, a function might accept a raw date string and return a standardized date format if valid.

✓ Remember, in Python, even if you don't explicitly return a value from a function, it still returns a value: `None`. So, in the context of Python, when we say a function "doesn't return a value," we generally mean it doesn't have a return statement with a meaningful value; instead, it implicitly returns **None**.



# why should we go for functions?

- 1. Modularity and Organization:** Functions allow you to break down large tasks or programs into smaller, more manageable pieces. This makes the code easier to read, understand, and debug.
- 2. Code Reusability:** Once a function is written, it can be called multiple times throughout a program. This prevents redundancy and means that you don't have to rewrite the same logic in multiple places.

# why should we go for functions?

- 3. Easier Maintenance:** When a task is encapsulated in a function, changes or fixes only need to be made in one location rather than everywhere the task occurs. This simplifies debugging and maintenance.
- 4. Abstraction:** Functions provide a layer of abstraction, letting you hide the complexity of what the function does. Callers don't need to know how the function works internally, just what it does.

# why should we go for functions?

- 5. **Code Portability:** A well-written function can often be used in different programs or projects, saving time and effort in future development.
- 6. **Flexibility:** As you expand or modify your program, functions can be easily enhanced or modified without affecting other parts of your program.

# Recursive function

- A function calls itself is called Recursive function
- You must have one termination or base condition and recursive condition
- When should you use recursion?
  - When the problem has a tree-like structure
  - When the problem requires backtracking

## Advantages of recursion

1. The code may be easier to write.
2. Extremely useful when applying the same solution.
3. Recursion reduce the length of code.

# Disadvantages of recursion

1. Recursive functions are generally slower than non-recursive function.
2. It may require a lot of memory space to hold intermediate results on the system stacks.
3. Hard to analyze or understand the code.
4. It is not more efficient in terms of space and time complexity.
5. The computer may run out of memory if the recursive calls are not properly checked.

# Recursive function

- A recursive function is one that calls itself to solve smaller parts of the same problem. It's often used in computer science to tackle problems that have repeating patterns.

# Where do we use recursive functions?

- **Problem-Specific Use:** Recursive techniques are particularly well-suited for problems that naturally exhibit recursive structures. For instance:
  1. **Tree or Graph Traversal:** Recursive approaches are frequently used for tasks like searching, inserting, or deleting in data structures like binary trees (e.g., in-order, pre-order, and post-order traversals). Similarly, depth-first search (DFS) in graphs can be implemented recursively.



# Where do we use recursive functions?

- **Problem-Specific Use:**

- 2. Divide-and-Conquer Algorithms:** Algorithms like merge sort or quicksort, divide the problem into smaller pieces, solve each piece, and then combine the results..
- 3. Backtracking:** Recursion is a natural fit for algorithms that need to explore multiple potential solutions, like the classic N-Queens puzzle or certain optimization problems.

# Recursive function programs

1. Factorial
2. Fibonacci Sequence
3. Calculate Power
4. Reverse a String
5. Sum of List

# Recursive function

- In a recursive function, you must have at least one base case (or termination condition) and a recursive case.

**1. Base case (or termination condition):** This prevents the recursion from going on indefinitely. It specifies when the recursion should stop and return a result without making a further recursive call. **Ensure there's always a base case to terminate the recursion.**

# Recursive function

**2. Recursive case:** This is where the function calls itself, typically with a modified argument. **Ensure the recursive function is always progressing toward reaching that base case.**

- Lastly, it's a good idea to visualize the recursion flow (at least initially) to grasp the process better and ensure correct implementation.

# Advantages of Recursive function

- Recursive functions are very useful specially when a problem can be naturally divided into smaller instances of the same problem
1. **Readability and Simplicity:** Python's syntax is clean and concise. When combined with the inherent elegance of recursive solutions, it often leads to more readable and straightforward code compared to iterative solutions.
  2. **Natural Expression of Problems:** Some problems are inherently recursive in nature, meaning they can be described in terms of themselves. In such cases, a recursive function is a natural fit, closely aligning with the problem's definition.

# Advantages of Recursive function

3. **Reduced Code Length:** In Python, recursive solutions can sometimes lead to much shorter code than their iterative counterparts.
4. **Built-in Data Structures:** Python has built-in support for data structures like lists and dictionaries, which can be easily integrated into recursive functions for tasks like Memoization.
  - ❖ For storing only non-negative integers, we use lists. Computing nth Factorial
  - ❖ If the problem requires to strings, tuples, even positive integers, we use Dictionaries

# Advantages of Recursive function

- 5. Standard Library Support:** Python's standard library provides tools that can be beneficial with recursion, such as the `functools.lru_cache` for automatic Memoization.

# Disadvantages of Recursive function

- **Stack Overflow:** Every time a function calls itself recursively, a new frame is added to the call stack. If the depth of the recursion is too deep, this can result in a stack overflow error. Python's default recursion limit is relatively low (usually 1000 recursive calls) to prevent this, but it can still be a problem for deep recursions.
- **Overhead of Function Calls:** Recursive functions often involve a series of function calls. Each call has an overhead associated with it, such as setting up the stack frame, which can make recursion slower than iterative solutions for some problems.



# Disadvantages of Recursive function

- **Memory Consumption:** Each recursive call uses some amount of memory, and if the recursion is deep, this can lead to substantial memory usage.
- **Understanding (Beginners):** For some programmers, especially those unfamiliar with recursion, recursive functions can be harder to understand than their iterative counterparts.

# Disadvantages of Recursive function

- **Debugging:** Debugging recursive functions can be more challenging, especially when trying to grasp the state at a particular level of recursion or when the recursion is deep.
- **Lack of Explicit Base Case:** If a base case is not explicitly defined or is defined incorrectly, the recursive function can result in infinite recursion, which will eventually lead to a stack overflow error or hang the program.

# Disadvantages of Recursive function

- Despite these disadvantages, recursive functions are extremely useful for certain types of problems, especially those that can naturally be broken down into smaller sub-problems (like tree traversals, certain divide-and-conquer algorithms, etc.). However, in Python, it's often advisable to be cautious with recursion, especially for functions that might be called with large input values.

# Memoization

- Memoization is an optimization technique used primarily for recursive algorithms. It involves storing the results of expensive function calls and returning the cached results when the same inputs are provided again. This avoids redundant computations and can significantly speed up the program.
- Both lists and dictionaries in Python can be effectively used for memoization in recursive functions:

# Memoization vs Memorization

| Attribute   | Memoization                                                                                                                                                                                             | Memorization                                                                                                                                              |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Domain      | Computer Science and Programming                                                                                                                                                                        | General , Education, Psychology                                                                                                                           |
| Definition  | Memoization is an optimization technique used primarily with recursive algorithms. It involves caching the results of expensive function calls to avoid redundant computations and improve performance. | Memorization refers to the process of committing information to memory or learning something by heart.                                                    |
| Application | Used in dynamic programming to store results of costly computations and retrieve them when needed, thereby reducing the computational complexity of algorithms.                                         | Used in everyday life and in educational contexts when someone needs to remember information, like when students memorize formulas, dates, or vocabulary. |

# Memoization vs Memorization

| Attribute | Memoization                                                                                                                                                     | Memorization                                                                                                             |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| Example   | Computing the Fibonacci sequence using memoization ensures that the Fibonacci number for a specific value is computed only once and then stored for future use. | Students might memorize multiplication tables in elementary school to quickly recall results without recalculating them. |

In Essence:

- **Memoization** is about saving computed results to avoid re-computation (in programming).
- **Memorization** is about storing and recalling information in one's memory (in learning).

# Caveats to be aware of when using recursion

- However, there are some caveats to be aware of when using recursion in Python:
- **Stack Limit:** Python's default recursion limit is relatively low (typically **1000 recursive calls**) to prevent a crash due to a stack overflow. If you know your recursion will be deeper than this and you're certain it won't cause a stack overflow, you can manually increase the recursion limit using `sys.setrecursionlimit()`. But caution is advised.

# Caveats to be aware of when using recursion

- Despite these concerns, when applied judiciously and for the right problems, recursive functions in Python can offer elegant, clear, and efficient solutions.



| Property        | Local Variables                                                                             | Global Variables                                                                                                                              |
|-----------------|---------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Scope</b>    | Local variables are defined inside a function and are only accessible within that function. | Global variables are defined outside of functions or module or script level blocks and are accessible throughout the entire module or script. |
| <b>Lifetime</b> | They are created when the function is called and destroyed when the function exits.         | They are created when their containing module or script starts and are destroyed when it ends.                                                |

# Python String Functions

- **Count()**: count is used to count the number of occurrences of a substring in string

## Syntax:

Strvar/value.count()

Takes minimum of 1 and maximum of 3 arguments

# Python String Functions

- **Capitalize():** This function make the first character have upper case and the rest lower case.
- **Title():** words start with uppercased characters and all remaining cased characters have lower case.
- **Istitle():** Return True if the string is a title-cased string, False otherwise.

# Python String Functions

- `Startswith()`: Return True if S starts with the specified prefix, False otherwise. prefix can also be a tuple of strings to try.
- If your **prefix** can start with more than one, then we should go with the tuple of strings

# Python String Functions

- `Endswith()`: Return True if S starts with the specified suffix, False otherwise. suffix can also be a tuple of strings to try.
- If your **ends** can start with more than one, then we should go with the tuple of strings

# Python String Functions

- `find()`: It is used to find the occurrence of the substring between the boundaries. If the sub string is present, it returns the lowest index of the sub string. Return **-1** on **failure**.

# Python String Functions

- `index()`: It is used to find the occurrence of the substring between the boundaries. If the sub string is present, it returns the lowest index of the sub string .  
Raises `ValueError` when the substring is not found.

## Find and index Real Time Scenarios

The choice between them depends on your preference for error handling and whether you want to handle the absence of the substring with custom exception handling (error messages) or by checking the return value (find\_and\_replace)



## Find and index Real Time Scenarios

Where the calling function asserts, we should use find rather index and also, we have seen in place someone has asserted '0' as false which is wrong as per find we should assert as  $\text{index} > -1$

# Python String Functions

- `isalpha(), isdigit(), isalnum()`
- `Upper(), lower()`
- `Isupper(), islower()`

# Python String Functions

- **Replace()**: Replace function is used to replace older substring in a string with new substring
- If the older substring is more than once, then it replace in all positions with new substring
- It is also possible to specify replacement count, to say how many number times to replace
- `Strvar/value.replace('OS','NS',count)`

# Python String Functions

- **Strip():** Removes leading and trailing whitespace characters from a string.

# Escape sequences

- In Python, the backslash ("**\**") is used as an **escape character** and it has a **special meaning** when you use with **the strings**.
- An escape sequence is a combination of a **backslash and a character** that represents a **special character or action**.
- Here are some common escape sequences in Python:
  - **\\: Backslash** - Represents a literal backslash character.
  - **\': Single Quote** - Represents a single quotation mark (apostrophe) within a single-quoted string.

# Escape sequences

- **\": Double Quote** - Represents a double quotation mark within a double-quoted string.
- **\n: Newline** - Represents a newline character, causing the text to move to the next line.
- **\t: Tab** - Represents a horizontal tab character, creating an indentation.
- **\b: Backspace** - Represents a backspace character, used to erase the previous character.

# Escape sequences

- **\r: Carriage Return** - Represents a carriage return character, which moves the cursor to the beginning of the line without advancing to the next line. Often used in combination with `\n` to represent Windows-style line endings.
- **\f: Form Feed** - Represents a form feed character, which is used for page breaks or formatting purposes.

# Python String Functions

- The `split()` function is used to split a string into a list of substrings based on a specified delimiter (e.g., a space, comma, or any character).
- `split()` is useful for breaking down a string into parts



# Python String Functions

```
>>> s='Hello Anilkumar How are you'
>>> s.split()
['Hello', 'Anilkumar', 'How', 'are', 'you']
>>> s='Hello Anilkumar\\n How are you'
>>> s.split()
['Hello', 'Anilkumar\\n', 'How', 'are', 'you']
>>> s='Hello Anilkumar\\nHow are you'
>>> s.split()
['Hello', 'Anilkumar', 'How', 'are', 'you']
```

```
>>> s='HelloHeroSuperHero'
>>> s.split('o')
['Hell', 'Her', 'SuperHer', '']
```

# Python String Functions

- Always the split words will be count+1
- As soon as split function starts it is going to open ‘
- If the split string is not given to split function, then by default it considers white space, escape characters (`\n`, `\r`, `\t`, `\f`, `\v`) as split string and divide string into smaller pieces
- If you have given a split string which is not available, it returns as it is in list format
- How split function works is it always starts (‘) writing the string from beginning and if it see split string stops writing and ends with (‘), then again ‘starts writing until it sees split string or ‘null’ characters

# Python String Functions

- `join()` is useful for merging a list of strings into a single string with a specified separator
- Only 3 examples for split and join with default, - and character

# Python String Functions

- These functions are frequently used when **working with text data** in Python. **split()** is useful for breaking down a string into parts, while **join()** is useful for merging a list of strings into a single string with a specified separator.

# Python String Functions

## **Syntax:**

`gluestring.join(a)` where `a` is iterable

Glue joins the two split strings like paper

# Python list Functions

- `append()` is used to add a single element to the end of a list.
- The argument passed to `append()` becomes a single element of the list, even if it's an iterable (e.g., a list or tuple).
- It does not `modify the original iterable` but adds the `entire iterable as a single element`.

# Python list Functions

- `extend()` is used to append elements from an iterable (e.g., another list, tuple, or string) to the end of an existing list.
- It takes an iterable as an argument and adds its individual elements to the list.
- It modifies the original list by adding multiple elements.

## Key diff b/w append and extend

- So, the key difference is that `append()` adds a single element to the end of the list, while `extend()` adds multiple elements from an iterable to the end of the list. The choice between them depends on your specific needs when working with lists in Python.



# Python list Functions

- **insert():** Inserts an element at a specific index in the list.
- When you insert the specified element, and the existing elements will be shifted 1 position to right to accommodate it.

# Python list Functions

- `pop()` is used to **remove and return** an element from a list based on its **index**.
- It **takes one argument**, which is the **index of the element** to be removed.
- The element at the specified index is removed from the list, and the method returns the removed element.
- If **no index is provided**, it removes and returns the **last element** of the **list** by default.
- If the index is not found in the list, **it raises a `IndexError`**.

# Python list Functions

- `remove()` is used to remove the first occurrence of a specified element from a list.
- It takes **one argument**, which is the element to be removed.
- It searches for the first occurrence of the specified element and removes it from the list.
- If the element is not found in the list, **it raises a ValueError**.

# Python list Functions

- `pop()` uses the index of the element to be removed.
- `remove()` uses the value of the element to search for and remove it from the list.
- Additionally, `pop()` returns the removed element, while `remove()` does not return anything; it just modifies the list in place.

# Python list Functions

- The `clear()` method in Python is used to remove all elements from a list, effectively making the list empty. It modifies the original list in place and does not return any value
- The `reverse()` method in Python is used to reverse the order of elements in a list. It modifies the original list in place and does not return any value.

# Python list Functions

- The `sort()` method in Python is used to sort the elements of a list in ascending order (from lowest to highest). It modifies the original list in place and does not return any value.

# Python list Functions

- The `count()` method in Python is used to count the number of occurrences of a specified element within a list. It returns the count of the specified element in the list.

# Python list Functions

- The `index()` method in Python is used to find the index (position) of the first occurrence of a specified element within a list. It returns the index of the specified element in the list.



# Python tuple Functions

- Python **tuples** are **immutable**, so we can't add/change/ remove the elements from the tuple
- So, tuple has only **2** functions **count()** and **index()**

# Python set Functions

- **Adding Elements:**
- `add(element)`: Adds an element to the set.
- `update(iterable)`: Adds multiple elements to the set.

# Python set Functions

- **Removing Elements:**

- `remove(element)`: Removes an element from the set. Raises an error if the element is not found.
- `discard(element)`: Removes an element from the set if it exists, but does not raise an error if the element is not found.
- `pop()`: Removes and returns an arbitrary element from the set.

# Python set Functions

- **Set Operations:**

- `union(other_set)`: Returns a new set containing all unique elements from both sets.
- `intersection(other_set)`: Returns a new set containing elements that are common to both sets.

# Python set Functions

- **Set Operations:**

- `difference(other_set)`: Returns a new set containing elements that are in the first set but not in the second set.
- `symmetric_difference(other_set)`: Returns a new set containing elements that are in either of the sets, but not in both.

# Python set Functions

- **Set Comparison:**
- `issubset(other_set)`: Checks if the set is a subset of another set.
- `issuperset(other_set)`: Checks if the set is a superset of another set.
- `Isdisjoint()` – indirectly checks `intersection()`

# Python set Functions

- **Copying and clearing sets:**
- `copy()`: Creates a shallow copy of the set.
- `clear()`: Removes all elements from the set, making it empty.

# Shallow copy vs deep copy

## Shallow copy

It copies only values

Points to **different** memory location

**Example:**

```
d1={'name':'anil'}  
d2=d1.copy()
```

## Deep copy

It copies both values and **address**

Points to **same** memory location

**Example:**

```
d1={'name':'anil'}  
d2=d1
```