Kakunje Software™

engineering ideas

INTRODUCTION TO PYTHON

About Python

- Python is a high-level, interpreted, object-oriented programming language.
- Python was developed by **Guido Van Rossum** in 1989 and was released in 1991.
- The name Python was taken from the popular BBC comedy show of the time, **“Monty Python’s Flying Circus”**.
- Support multiple programming paradigms (procedural, object-oriented, functional).

Features of Python

Easy to learn and use

Cross platform
(Platform Independent)

Free and open
source

Large Standard Library

Object oriented
language

Memory
Management

Uses of Python

Web Applications

Machine learning

Game Developments

Artificial Intelligence

Desktop Applications

Data Sciences

Internet of things

Embedded system

Automation

Python Development Tools and IDE’S

IDE

Jupyter Notebook

VS Code

Google Colab

PyCharm

Thonny IDE

IDLE

PYTHON IDLE INSTALLATION:

- Go to Python website(https://www.python.org/).
- Click on Downloads → Windows → **Download Python 3.12.0** (Windows x86-64 executable installer.
- Run the downloaded installer → Check the box **“Add Python to PATH”** → Click **“Customize Installation”** → Ensure **IDLE** is checked → Select **“Install for all users”** → Click **“Install Now”** and wait for the installation to complete. Once installed, click **“Close”**.
- After installation go to the taskbar and click on the search bar and type **“IDLE”**. Right click on **IDLE (Python 3.12 64-bit)** and select “**Open file Location”** and copy its **path address**.
- Open **Command Prompt** and select “**Run as administrator”** → Click **“Yes”** → Type **cd** and paste the copied path and press **enter**.
- Install libraries / packages using pip

VS CODE:
Visual Studio Code (VS Code) is a free, cross-platform source-code editor developed by Microsoft

VS CODE INSTALLATION:

- Go to Visual Studio Code website(https://code.visualstudio.com). On the homepage, click the **"Download for Windows/Mac/Linux"** button.
- Run the downloaded **.exe** file → Select **"I accept the agreement"** → Click **Next** → Click **Next** again → Check the option **"Add to PATH"** → Click **Next** → Click **Install**, then **Finish** after the installation is complete.
- Open **VS Code** from your **desktop** or **application menu** → Click on **Extension** tab and search for **Python** and install the **Python Extension**.
- In the **VS Code** go to the search bar and type **"Show and Run Commands"** and select it. Choose **"Python: Select Interpreter"** and choose the installed **Python version (IDLE interpreter)**.

JUPYTER NOTEBOOK INSTALLATION:

- Go to the official Anaconda website and download the Anaconda Individual Edition for your operating system.

- Run the installer and follow the on-screen instructions. During the installation, make sure to check the option **“Add Anaconda to my PATH environment variable”**.

- After installation is complete, open the Anaconda Navigator application. In the home tab you should see Jupyter Notebook as an available option. Click the **Launch** button next to it. This will open Jupyter Notebook in your default web browser.

- Once Jupyter Notebook opens in your browser, click on the **New** button (top-right corner), then select **Python 3** (or another kernel if you have installed different versions). You can now begin writing and running Python code in the notebook cells.

GOOGLE COLAB

Google Colab is a **cloud-based Jupyter Notebook environment** provided by Google that allows users to write and execute Python code directly in a web browser without needing any setup. It is widely used for machine learning, deep learning, data analysis, and Python programming.

GOOGLE COLAB INSTALLATION

- Go to Google Colab website (https://colab.research.google.com)
- Sign in with your Google account if prompted.
- Click on **"File"** → **"New Notebook"** to create a new Python notebook. A new Colab notebook (.ipynb) will open with a code cell.
- Click on **"Untitled"** at the top-left and rename your notebook.
- Click inside a code cell and type your Python code.

PYCHARM

PyCharm is a popular **Integrated Development Environment (IDE)** for Python, developed by JetBrains.

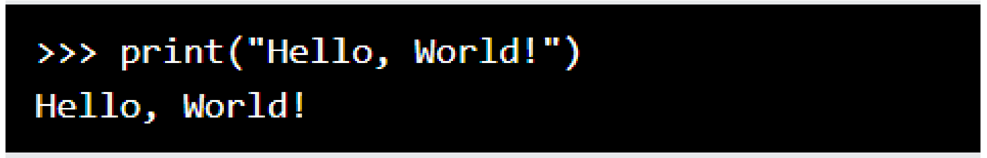
PYCHARM INSTALLATION

- Go to the official website (https://www.jetbrains.com/pycharm)
- Choose your OS: Windows, macOS, or Linux
- Download the **Community Edition** (free) or **Professional Edition** (paid).
- Run the downloaded **.exe** file.
- Click **Next** and choose the installation path.
- Select options: **“Add PyCharm to PATH”** and **“Associate .py files with PyCharm”**.
- Click **Install**, then **Finish** after completion.
- Launch PyCharm and go to **File** → **Settings** → **Project Interpreter** and select **Python Interpreter**.

Getting started with Python Programming

Print Hello World:

- Python single line codes can be executed directly on the command line.



```
>>> print("Hello, World!")
Hello, World!
```

- To execute multiple line codes, open a text editor(Python IDLE, VS Code, PyCharm, etc.) and write your Python Code. Save the file with a **.py** extension.

PYTHON SYNTAX

PYTHON INDENTATION

- Python Indentation refers to the use of whitespace (spaces or tabs) at the beginning of code line.
- It is used to define the code blocks.
- Indentation is crucial in Python because, unlike many other programming languages that use braces "{}" to define blocks, Python uses indentation.
- It improves the readability of Python code, but on other hand it became difficult to rectify indentation errors.
- Even one extra or less space can lead to indentation error.

Example

```
if 10 > 5:
    print("This is true!")
    print("I am tab indentation")
print("I have no indentation")
```

Syntax Error

```
if 5 > 2:
print("Five is greater than two!")
```

PYTHON KEYWORDS

- Keywords in Python are reserved words that have special meanings and serve specific purposes in the language syntax.
- They cannot be used as identifiers (names for variables, functions, classes, etc.). For example, "for", "while", "if", and "else" are keywords and cannot be used as identifiers.
- Below is the list of keywords in Python

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

PYTHON COMMENTS

- Comments in Python are statements written within the code.
- They are meant to explain, clarify, or give context about specific parts of the code.
- The purpose of comments is to explain the working of a code, they have no impact on the execution or outcome of a program.

Python Single Line Comment

- Single line comments are preceded by the "#" symbol.
- Everything after this symbol on the same line is considered as a comment.

Example

```
first_name = “Liam"
last_name = “Refsnes"  # assign last name

# print full name
print(first_name, last_name)
```

Python Multi-line Comments:

- Python doesn't have a specific syntax for multi-line comments.
- However, programmers often use multiple single-line comments, one after the other, or sometimes triple quotes (either ''' or """), even though they're technically string literals. Below is the example of multiline comment.

Example 1

```
#This is a comment written in
#more than just one line
print(“Hello World!”)
```

Example 2

```
“ “ “
This is a comment written in
more than just one line
” ” ”
print(“How are you?”)
```

CONTINUATION OF STATEMENTS IN PYTHON

- In Python, statements are typically written on a single line.
- However, there are scenarios where writing a statement on multiple lines can improve readability or is required due to the length of the statement.
- This continuation of statements over multiple lines is supported in Python in various ways:

Implicit Continuation

- Python implicitly supports line continuation within parentheses (), square brackets [], and curly braces {}.
- This is often used in defining multi-line lists, tuples, dictionaries, or function arguments.

Example

```
# Line continuation within square brackets '[]'
numbers = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
]
print(numbers)
```

Explicit Continuation

- You can use backslash '\' to indicate that a statement should continue on the next line.

Example

```
# Explicit continuation
s = “Python is awesome! " \
   “It is easy to learn!"

print(s)
```

Note: Using a backslash does have some pitfalls, such as if there's a space after the backslash, it will result in a syntax error.

TAKING INPUT FROM USER IN PYTHON

- The **input()** function in Python is used to take user input from the console.
- The program execution halts until the user provides input and presses "Enter".
- The entered data is then returned as a string.
- We can also provide an optional prompt as an argument to guide the user on what to input.

Example : In this example, the user will see the message "Please enter your name: ". After entering their name and pressing "Enter", they'll receive a greeting with the name they provided.

```
# Taking input from the user
name = input("Please enter your name: ")

# Print the input
print("Hello," + name)

age = int(input(“Please enter your age: “))

Print(“Your age is “, age)
```

PYTHON VARIABLE

PYTHON VARIABLES

- In Python, variables are used to store data that can be referenced and manipulated during program execution.
- A variable is essentially a name that is assigned to a value.
- In other programming languages, Python variables do not require explicit declaration of type. The type of the variable is inferred based on the value assigned.

Example

```
 # Variable 'x' stores the integer value 10
x = 5
# Variable 'name' stores the string "Samantha"
name = "Samantha"

print(x)
print(name)
```

RULES FOR NAMING VARIABLES

- A variable name cannot start with a number.
- A variable name can only contain letters, digits and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (date, Date and DATE are three different variables)
- There are several techniques to write multi words variables
  * Camel case (myVariableName)
  * Pascal case (MyVariableName)
  * Snake case (my_variable_name)

Example

```
age = 21
_color = "lilac"
total_score = 90
```

MULTIPLE ASSIGNMENTS

- Python allows multiple variables to be assigned values in a single line.

Assigning the Same Value

- Python allows assigning the same value to multiple variables in a single line, which can be useful for initializing variables with the same value.

Example

```
a = b = c = 100
print(a, b, c)
```

Assigning Different Values

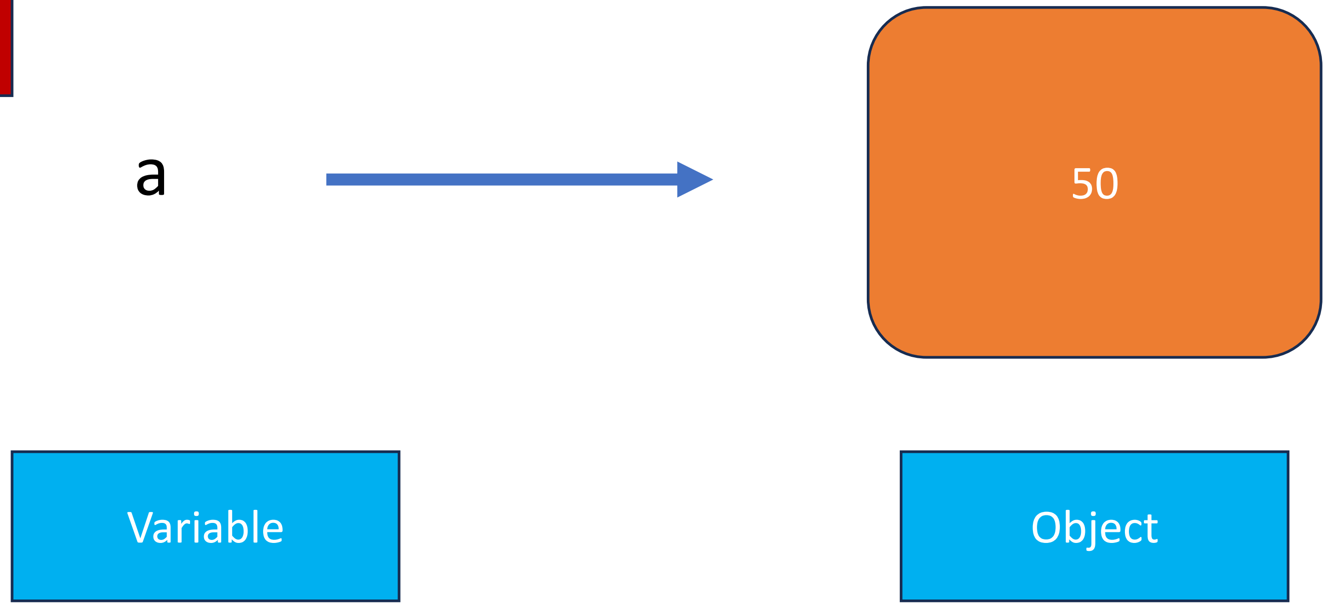
- We can assign different values to multiple variables simultaneously, making the code concise and easier to read.

Example

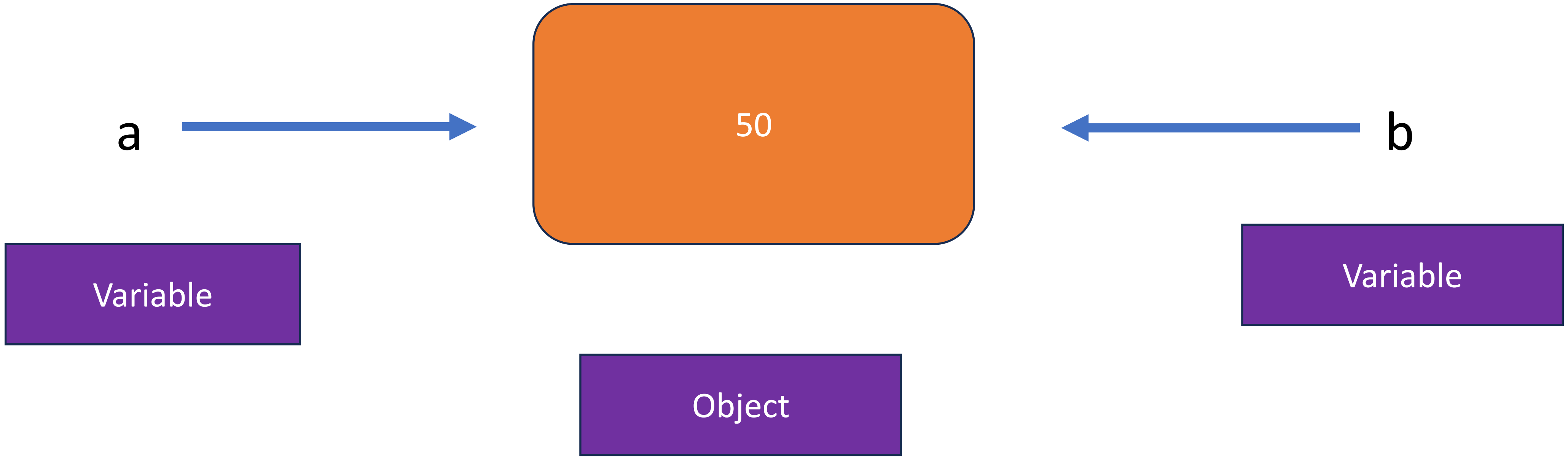
```
x, y, z = 1, 2.5, "Python"
print(x, y, z)
```

Object References

a = 50



```
a = 50
b = a
```



GETTING THE TYPE OF VARIABLE

- In Python, we can determine the type of a variable using the **type()** function.
- This built-in function returns the type of the object passed to it..

Example

```
x = 5
y = “Ram”
print(type(x))
print(type(y))
```

PYTHON DATA TYPES

NUMERIC TYPES
- int
- float
- complex

TEXT TYPE
- str

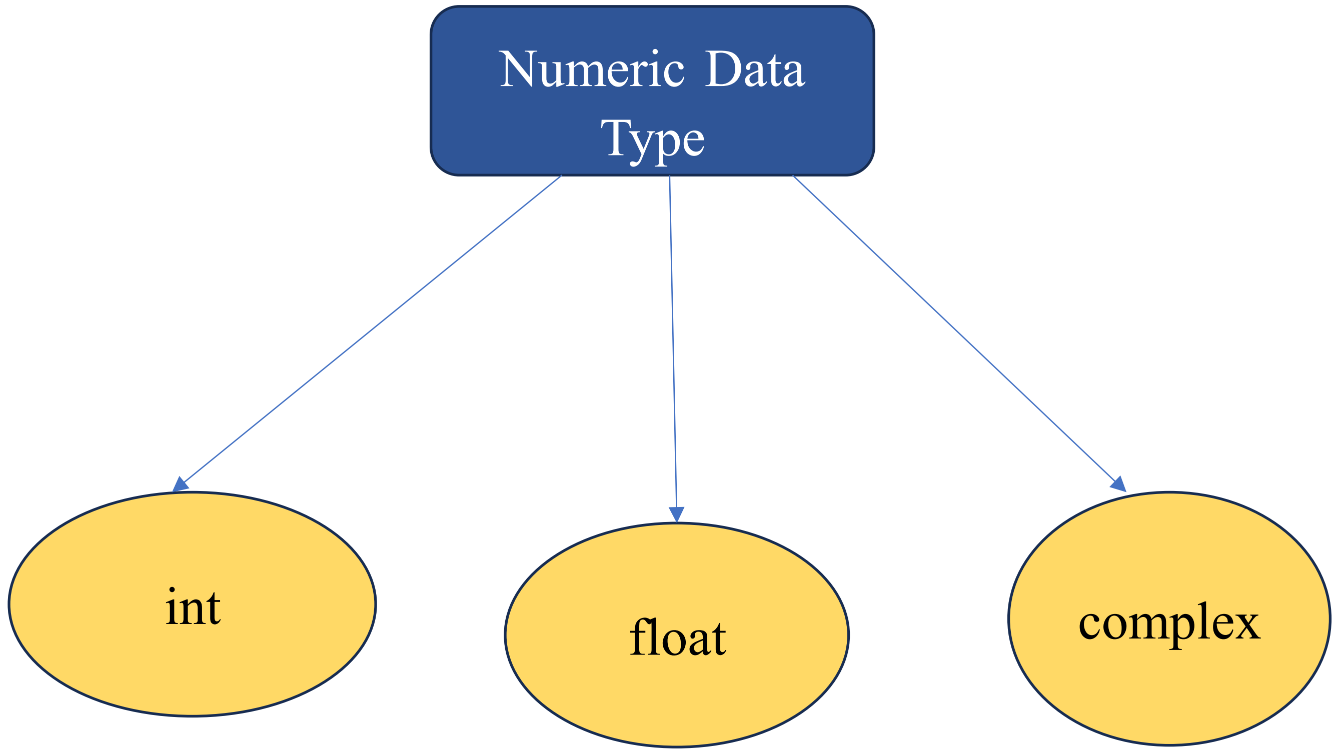
SEQUENCE TYPES
- List
- Tuple

SET TYPE
- Set

MAPPING TYPE
- Dictionary

BOOLEAN TYPE
- bool

NUMERIC DATA TYPE



int

int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

```
x = 1
y = 3569871122556364428
z = -8955641

print(x, y, z)
```

float

float, or floating point number is a number, positive or negative, containing one or more decimals. Float can also be scientific numbers with an **"e"** to indicate the power of 10.

Example

```
x = 50.6987
y = 6.8
z = -56.587
a = 35e3
b = 12E7
c = -97.9e10

print(x, y, z)
print(a, b, c)
```

complex

complex numbers are written with a **“j”** as the imaginary part.

Example

```
x = 3 + 5j
y = 9j
z = -8j

print(x)
print(y)
print(z)
```

Type Conversion

You can convert from one type to another with the int( ), float( ), and complex( ) methods.
You cannot convert complex numbers into another number type.

Example

```
 x = 1
 y = 2.8
 z = 1j
#convert from int to float
 a = float(x)
#convert from float to int
 b = int(y)
#convert from int to complex
 c = complex(x)

print(a)         #1.0
 print(b)        #2
 print(c)        #(1+0j)

 print(type(a)) #<class ‘float’>
 print(type(b)) #<class ‘int’>
 print(type(c)) #<class ‘complex’>
```

TEXT TYPE -STRING

- A string is a sequence of characters.
- Python treats anything inside quotes as a string.
- This includes letters, numbers, and symbols.
- Python has no character data type so single character is a string of length 1.

Strings can be defined inside a single quote or double quotes.

Using Single quote
str1 = ‘Python’

Using Double quotes
str2 = “Hello, How are you?”

Triple quotes are generally used for multi-line string.

```
str3 = ‘ ‘ ‘ Python is a
           simple programming
           language ’ ’ ’
```

String Indexing

REVERSE INDEX

[-11]	[-10]	[-9]	[-8]	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]
H	E	L	L	O		W	O	R	L	D
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

FORWARD INDEX

ACCESSING CHARACTERS IN STRING

- Strings in Python are arrays of bytes representing Unicode characters.
- 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.

Example

```
str = “Hello World”
print(str[1])
print(str[9])
print(str[-3])
print(str[-5])
```

STRING LENGTH

In Python, we use len() function to return the length of a string.

Example

```
str1 = “Hello World”
print(len(str1))          # 11
str2 = “a”
print(len(str2))          # 1
str3 = “How are you?”
print(len(str3))          #12
```

STRING SLICING

You can return a range of characters by using the slice syntax.
Specify the start index and the end index, separated by a colon, to return a part of the string.

Example

```
 str = “Hello World”
 print(str[2:8])
# llo Wo
```

Slice from the Start

By leaving out the start index, the range will start at the first character.
Example: Get the characters from the start to position 5.

```
 str = “Python is awesome”
 print(str[:5])                    # Pytho
```

Slice to the End

By leaving out the *end* index, the range will go to the end.
Example: Get the characters from position 8, and all the way to the end.

```
 str = “Python is awesome”
 print(str[8:])                    # s awesome
```

NEGATIVE INDEXING

- Use negative indexes to start the slice from the end of the string.

Example:

```
 str = “Hello world”
 print(str[-5:-2])
# wor
```

STRING CONCATENATION

- To concatenate, or combine, two strings you can use the + operator.

Example: Merge variable ‘a’ with variable ‘b’ into variable ‘c’.

```
a = “Hello”
b = “World”
c = a + b
print(c)        #HelloWorld
```

To add a space between them, and a “ ”

```
c = a + “ ” + b
print(c)        #Hello World
```

Note: We cannot combine strings and numbers using + symbol.

Example FORMAT ERROR

```
age = 20
txt = “My name is Alex, I am ” + age
print(txt)
```

STRING FORMAT

To format the string we use f-string.
To specify a string as an f-string, simply put an ‘f’ in front of the string literal, and add curly brackets {} as placeholders for variables and other operations.

Example

```
age = 20
txt = f“My name is Ram, I am {age}”
print(txt)              # My name is Ram, I am 20
```

Example:

Add a placeholder for the price variable:

```
price = 89
txt = f“ The price is {price} dollars”
print(txt)                    # The price is 89 dollars
```

A placeholder can contain Python code, like math operations.

Example:

Perform a math operation in the placeholder, and return the result.

```
txt = f“The price is {20 + 30} dollars”

print(txt)                    #The price is 50 dollars
```

A placeholder can include a modifier to format the value.
A modifier is included by adding a colon : followed by a legal formatting type, like .2f which means fixed point number with 2 decimals.

Example

```
price = 30
txt = f“The price is {price:.2f} dollars”
print(txt)              # The price is 30.00 dollars
```

ESCAPE CHARACTER

- To insert characters that are illegal in a string, use an escape character.
- An escape character is a backslash \ followed by the character you want to insert.

Example

```
txt = “We are the so called “Vikings” from the north.”
# You will get an error if you use double quotes inside a string
that is surrounded by double quotes.
```

To fix this problem, use the escape character \”

```
txt = “We are the so called \“Vikings\” from the north.”
print(txt)
# We are the so called “Vikings” from the north.
```

STRING METHODS

- Python has a set of built-in methods that you can use on strings.
- All string methods return new values. They do not change the original string.

Some of the string methods are:

❖**Upper Case**: The **upper()** method returns the string in upper case.

```
a = “Hello world”
print(a.upper())         #HELLO WORLD
```

❖**Lower Case:** The **lower()** method returns the string in lower case.

```
a = “HELLO WORLD”
print(a.lower())         # hello world
```

❖**Remove Whitespace:**

- Whitespace is the space before and/or after the actual text, and very often you want to remove this space.
- The **strip()** method removes any whitespace from the beginning or the end.

```
a = "   Hello world    "
print(a.strip())           #Hello world
```

❖**Replace string:** The **replace()** method replaces a string with another string.

```
a = "MOM"
print(a.replace("M", "D"))       #DOD
```

❖**Split string:** The **split()** method returns a list where the text between the specified separator becomes the list items.

```
txt = "Python is awesome"
print(txt.split(" "))                #['Python', 'is', 'awesome']
```

SEQUENCE TYPES

LIST

Definition:-

A list is a collection of objects, values, or items of different types and these collections are enclosed within the square brackets **[ ]** and separated by commas (**,**).

Example: -

```
list1 = [“apple”, 50, True, 40.569, “car”]
```

List items are indexed, the first item has index [0], the second item has index [1], etc. It also supports the reverse indexing, the last item has index [-1] and so on.

Characteristics Of List

- The lists are **ordered.** This means that the items have a defined order, and that order will not change. (**Note: -** There are some list methods that will change the order, but in general, the order of the items will not change.)
- List can store different types of elements.
- Elements of the list can be accessed by index.
- The lists are **mutable** or **changeable.** This means that we can change, add, and remove items in a list after it has been created.
- Lists allow duplicate elements.

LIST LENGTH: -
To determine how many items a list has, use the **len()** function.

Example

```
list1 = [10, 20, 30, 40, 50]
print(len(list1))                    #5
```

type( ) – Lists are defined as objects with the data type ‘list’.

Example

```
list2 = [“apple”, 50, True, 40.569, “car”]
print(type(list2))                          #<class  ‘list’>
```

Access List Items

- List items are indexed and you can access them by referring to the index number.

Example

```
myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
print(myList[3])          #grapes
print(myList[0])          #apple
print(myList[-4])         #banana
print(myList[-1])         #mango
```

Range of Indexes: -

- You can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new list with the specified items

Example

```
 myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
print(myList[1:4])       #[‘banana’, ‘orange’, ‘grapes’]
print(myList[:3])        #[‘apple’, ‘banana’, ‘orange’]
print(myList[2:])        #[‘orange’, ‘grapes’, ‘mango’]
print(myList[-4:-2])     #[‘banana’, ‘orange’]
```

Check if Item Exists or not: -

```
myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
if “orange” in myList:
	print(“Yes, ‘orange’ is in this list.”)
else:
	print(“No, ‘orange’ is not in this list.”)
#Yes, ‘orange’ is in this list.

if “orange” not in myList:
	print(“Yes, ‘orange’ is not in this list.”)
else:
	print(“No, ‘orange’ is in this list.”)
#No, ‘orange’ is in this list.
```

Change List Items

CHANGE ITEM VALUE

- To change the value of a specific item, refer to the index number.

Example

```
myList = [“apple”, “banana”, “cherry”]
myList[1] = “mango”
print(myList)
#[‘apple’, ‘mango’, ‘cherry’]
```

Change a Range of Item Values:-

- To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example

```
 myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
myList[1:3] = [“kiwi”, “cherry”]
print(myList)
#[‘apple’, ‘kiwi’, ‘cherry’, ‘grapes’, ‘mango’]
```

- If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly.

Example

```
 myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
myList[1:2] = [“kiwi”, “cherry”]
print(myList)
#[‘apple’, ‘kiwi’, ‘cherry’, ‘orange’, ‘grapes’, ‘mango’]
```

The length of the list will change when the number of items inserted does not match the number of items replaced.
If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly.

Example

```
myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
myList[1:3] = [“kiwi”]
print(myList)
#[‘apple’, ‘kiwi’, ‘grapes’, ‘mango’]
```

Add List Items

Insert Items:

- To insert a new list item, without replacing any of the existing values, we can use the **insert( )** method.
- The insert( ) method inserts an item at the specified index.

Example

```
myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
myList.insert(2, “cherry”)
print(myList)
#[‘apple’, ‘banana’, ‘cherry’, ‘orange’, ‘grapes’, ‘mango’]
```

Note: - The list will now contain 6 items.

Append Items

- To add an item to the end of the list, use the **append()** method.

Example

```
myList = [“apple”, “banana”, “orange”, “grapes”, “mango”]
myList.append(“cherry”)
print(myList)
#[‘apple’, ‘banana’, ‘orange’, ‘grapes’, ‘mango’, ‘cherry’]
```

Extend List Items

- To append elements from another list to the current list, use the **extend()** method.

Example

```
list1 = [“apple”, “banana”, “orange”]
list2 = [“grapes”, “mango”, “cherry”]
list1.extend(list2)
print(list1)
#[‘apple’, ‘banana’, ‘orange’, ‘grapes’, ‘mango’, ‘cherry’]
```

Remove List Items

Remove Specified Item

The **remove()** method removes the specified item.

Example

```
list1 = [“apple”, “banana”, “cherry”]
list1.remove(“banana”)
print(list1)
#[‘apple’, ‘cherry’]
```

Note: - If there are more than one item with the specified value, the **remove()** method removes the first occurrence.

Example

```
list1 = [“apple”, “banana”, “cherry”, “banana”, “mango”]
list1.remove(“banana”)
print(list1)
#[‘apple’, ‘cherry’, ‘banana’, ‘mango’]
```

Remove Specified Index

The **pop()** method removes the specified index.

<u>Example: -</u>

```
list1 = [“apple”, “banana”, “cherry”]
list1.pop(1)
print(list1)
#[‘apple’, ‘cherry’]
```

Note: - If you do not specify the index, the pop() method removes the last item.

<u>Example: -</u>

```
list1 = [“apple”, “banana”, “cherry”]
list1.pop()
print(list1)
#[‘apple’, ‘banana’]
```

Note: - The **del** keyword also removes the specified index.

Example: -

```
list1 = [“apple”, “banana”, “cherry”]
del list1[2]
print(list1)
#[‘apple’, ‘banana’]
```

Note: - The **del** keyword can also delete the list completely.

Example: -

```
list1 = [“apple”, “banana”, “cherry”]
del list1
print(list1)
#list1 = [“apple”, “banana”, “cherry”]
#del list1
#print(list1)
#NameError: name ‘list1’ is not defined
```

Clear the List

The **clear()** method empties the list.

The list still remains, but it has no content.

Example: -

```
list1 = [“apple”, “banana”, “cherry”]
list1.clear()
print(list1)
#[]
```

Sort Lists

List objects have a **sort()** method that will sort the list alphanumerically, ascending, by default.

Example

```
list1 = [“orange”, “mango”, “kiwi”, “pineapple”, “banana”]
list1.sort()
print(list1)
#[‘banana’, ‘kiwi’, ‘mango’, ‘orange’, ‘pineapple’]
```

Example

```
list2 = [100, 50, 65, 82, 23]
list2.sort()
print(list2)
#[23, 50, 65, 82, 100]
```

Sort Descending: -

To sort descending, use the keyword argument **reverse = True.**

Example

```
list1 = [“orange”, “mango”, “kiwi”, “pineapple”, “banana”]
list1.sort(reverse = True)
print(list1)
#[‘pineapple’, ‘orange’, ‘mango’, ‘kiwi’, ‘banana’]
```

Example

```
list2 = [100, 50, 65, 82, 23]
list2.sort(reverse = True)
print(list2)
#[100, 82, 65, 50, 23]
```

Reverse Order

The **reverse ()** method reverses the current sorting order of the elements.

Example

```
list1 = [“orange”, “mango”, “kiwi”, “pineapple”, “banana”]
list1.reverse()
print(list1)
#[‘banana’, ‘pineapple’, ‘kiwi’, ‘mango’, ‘orange’]
```

Copy Lists

You cannot copy a list simply by typing **list2 = list1**, because, list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.
You can use the built in List method **copy()** to copy a list.

Example

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
#['apple', 'banana', 'cherry']
```

Join Lists

Two or more lists in Python are joined or concatenated using **+** operator.

Example

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
#['a', 'b', 'c', 1, 2, 3]
```

To create 2- dimensional array:

Creating a two-dimensional array using nested lists

Example

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

Access the Elements of an Array

Accessing the elements of the first row

Example

```
two_dim_array=[[1,2,3],[4,5,6],[7,8,9]]
print(two_dim_array[0])                    #[1, 2, 3]
```

Accessing the elements of the first column

Example

```
two_dim_array=[[1,2,3],[4,5,6],[7,8,9]]
first_column = [row[0] for row in two_dim_array]
print(first_column)                        #[1, 4, 7]
```

Matrix Indexing

Matrix indexing in Python involves working with lists of lists. This approach allows you to create, access, and manipulate elements within a two-dimensional array (matrix) using standard Python list operations.

Basic Matrix Creation:

A matrix can be represented as a list of lists, where each inner list represents a row.

matrix = [[1,2,3],[4,5,6],[7,8,9]]

Accessing Elements:

To access an element in the matrix, you specify the row and column indices. Note that Python uses zero-based indexing.

Access element in the first row, second column

element = matrix[0][1]

print(element)# Output: 2

```
# Access the entire first row
first_row = matrix[0]  # Output: [1, 2, 3]

# Access the entire first column
first_column = [row[0] for row in matrix]  # Output: [1, 4, 7]
```

Slicing Rows and Columns:

You can use slicing to access submatrices or parts of rows and columns.

```
# Access the first two rows
first_two_rows = matrix[:2]  # Output: [[1, 2, 3], [4, 5, 6]]

# Access the first two columns
first_two_columns = [row[:2] for row in matrix]  # Output: [[1, 2], [4, 5], [7, 8]]
```

Modifying Elements:
You can modify elements by directly assigning a new value to a specific index.

```
# Change the element in the second row, third column to 10
matrix[1][2] = 10  # matrix now becomes [[1, 2, 3], [4, 5, 10], [7, 8, 9]]
```

Adding Rows and Columns:
To add a row, simply append a new list to the matrix. To add a column, you need to append elements to each existing row.

```
# Add a new row
matrix.append([10, 11, 12])  # matrix now becomes [[1, 2, 3], [4, 5, 10], [7, 8, 9], [10, 11, 12]]

# Add a new column
for row in matrix:
    row.append(5)  # matrix now becomes [[1, 2, 3, 5], [4, 5, 10, 5], [7, 8, 9, 5], [10, 11, 12, 5]]
```

Removing Rows and Columns:

To remove a row, use the `del` statement. To remove a column, you need to delete elements from each row.

```
# Remove the last row
del matrix[-1]  # matrix now becomes [[1, 2, 3, 5], [4, 5, 10, 5], [7, 8, 9, 5]]

# Remove the last column
for row in matrix:
    del row[-1]  # matrix now becomes [[1, 2, 3], [4, 5, 10], [7, 8, 9]]
```

Accessing the diagonal elements:

To access the diagonal elements of a matrix (elements where the row index equals the column index), you can use a simple loop to iterate over the rows and columns simultaneously.

Example:

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

diagonal_elements = [matrix[i][i] for i in range(len(matrix))]

print(diagonal_elements)  # Output: [1, 5, 9]
```

In this code:
- `matrix[i][i]` accesses the diagonal elements, where `i` represents both the row and column index because we are accessing the diagonal elements.
- `range(len(matrix))` generates a sequence of indices from 0 to the length of the matrix (exclusive), which allows us to iterate over each row and column index simultaneously.

Tuples

Definition:-

A tuple is a collection of objects, values, or items of different types and these collections are enclosed within the round brackets **( )** and separated by commas (**,**).

Example: -

```
tup1 = (“apple”, 1, “banana”, “cherry”)
```

Tuple items are indexed, the first item has index [0], the second item has index [1], etc. It also supports the reverse indexing, the last item has index [-1] and so on.

Characteristics Of Tuples

- The tuples are **ordered.** This means that the items have a defined order, and that order will not change.
- Tuples can store different types of elements.
- Elements of the tuples can be accessed by index.
- The tuples are **immutable** or **unchangeable.** This means that we cannot change, add, and remove items after the tuple has been created.
- Tuples allow duplicate elements.

TUPLE LENGTH: - To determine how many items a tuple has, use the **len()** function.

<u>Example</u>

```
tup1 = (10, 20, 30, 40, 50)
print(len(tup1))                    #5
```

type() – Tuples are defined as objects with the data type ‘tuple’.

<u>Example</u>

```
 tup1 = (10, 20, 30, 40, 50)
print(type(tup1))                   #<class ‘tuple’>
```

CREATE TUPLE WITH ONE ITEM: -

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

<u>Example</u>

```
tup1 = (“apple”,)
print(type(tup1))           #<class ‘tuple’>
```

<u>Example</u>

```
tup2 = (100)
print(type(tup2))           #<class ‘int’>
```

Access Tuple Items

- Tuple items are indexed and you can access them by referring to the index number.

Example

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])                         #banana
print(thistuple[-3])                        #apple
```

Range of Indexes: -

- You can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new tuple with the specified items.

Example

```
myTuple = (“apple”, “banana”, “orange”, “grapes”, “mango”)
print(myTuple[1:4])             #(‘banana’, ‘orange’, ‘grapes’)
print(myTuple[:3])              #(‘apple’, ‘banana’, ‘orange’)
print(myTuple[2:])              #(‘orange’, ‘grapes’, ‘mango’)
print(myTuple[-4:-2])                #(‘banana’, ‘orange’)
```

Update Tuple

- Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable**.
- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)                    #('apple', 'kiwi', 'Cherry')
```

Add Items: -

Since tuples are immutable, they do not have a built-in **append()** method, but there are other ways to add items to a tuple.

1. Convert into a list:

Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
print(thistuple)              #('apple', 'banana', 'cherry', 'orange')
```

2. Add Tuple to a Tuple:-

You are allowed to add tuples to tuples, so if you want to add one item, or many, create a new tuple with the item(s), and add it to the existing tuple.

Example

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple = thistuple + y
print(thistuple)              #('apple', 'banana', 'cherry', 'orange')
```

REMOVE ITEMS

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items.

Example

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
print(thistuple)          #('banana', 'cherry')
```

The **del** keyword can delete the tuple completely.

Example

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple)

#This will raise an error because the tuple no longer exists.
```

Unpack Tuples

When we create a tuple, we normally assign values to it. This is called "packing" a tuple.
In Python, we are also allowed to extract the values back into variables. This is called “unpacking”.

Example

```
fruits = (“apple”, “banana”, “cherry”)
(green, yellow, red) =fruits
print(green)            #apple
print(yellow)           #banana
print(red)          #cherry
```

The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Using Asterisk *

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list.

Example

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits
print(green)                #apple
print(yellow)               #banana
print(red)          #[‘cherry’, ‘strawberry’, ‘raspberry’]
```

Join Tuples

To join two or more tuples, you can use the **+** operator.

Example

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
#('a', 'b', 'c', 1, 2, 3)
```

Multiply Tuples: -

If you want to multiply the content of a tuple a given number of times, you can use the * operator.

Example

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
#('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

Python Sets

Definition:-

A set is a collection of objects, values, or items of different types and these collections are enclosed within the curly brackets **{ }** and separated by commas (**,**).

Example: -

```
set1 = {“apple”, 1, “banana”, “cherry”}
```

Characteristics Of Sets

- **Unordered** means that the items in a set do not have a defined order.
- Set can store different types of elements.
- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Set items are **changeable**, meaning that we cannot change the items after the set has been created. Once a set is created, you cannot change its items, but you can remove items and add new items.
- Set do not allow duplicate values.

LENGTH OF A SET: - To determine how many items a set has, use the **len()** function.

Example

```
set1 = {“apple”, “banana”, “cherry”}
print(len(set1))                #3
```

type() – Sets are defined as objects with the data type ‘set’.

Example

```
set1 = {“apple”, “banana”, “cherry”}
print(type(set1))               #<class ‘set’>
```

Access Set Items

- You cannot access items in a set by referring to an index or a key.
- 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.

Example

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x)
#apple
#banana
#cherry
```

Example

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
#True
```

Check if “apple” is NOT present in the set

Example

```
thisset = {"apple", "banana", "cherry"}
print(“apple" not in thisset)
#False
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

ADD SET ITEMS

To add one item to a set, use the **add()** method.

Example

```
thisset = {"apple", "banana", "cherry"}
thisset.add(“orange”)
print(thisset)
#{‘apple’, ‘orange’, ‘cherry’, ‘banana’}
```

Add Sets: -

To add items from another set into the current set, use the **update()** method.

Example

```
set1 = {“apple”, “banana”, “cherry”}
set2 = {“pineapple”, “mango”, “papaya”}
set1.update(set2)
print(set1)
#{‘pineapple’, ‘cherry’, ‘papaya’, ‘mango’, ‘apple’, ‘banana’}
```

The object in the **update()** method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries, etc.).

Example

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)
#{‘banana’, ‘cherry’, ‘apple’, ‘orange’, ‘kiwi’}
```

Remove Set Items

To remove an item in a set, use the **remove()**, or the **discard()** method.

Example

```
set1 = {“apple”, “banana”, “cherry”}
set1.remove(“banana”)
print(set1)
#{‘apple’, ‘cherry’}
```

If the item to remove does not exists, **remove()** will raise an error.

If the item to remove does not exist, **discard()** will **not** raise an error.

<u>Example</u>

```
set1 = {“apple”, “banana”, “cherry”}
set1.discard(“banana”)
print(set1)
#{‘apple’, ‘cherry’}
set1.discard(“mango”)
print(set1)
#{‘apple’, ‘cherry’}
```

pop(): -You can also use the **pop()** method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.

<u>Example</u>

```
set1 = {“apple”, “banana”, “cherry”}
x = set1.pop()
print(x)
print(set1)
#banana
#{‘apple’, ‘cherry’}
```

clear(): -The **clear()** method empties the set.

<u>Example</u>

```
set1 = {“apple”, “banana”, “cherry”}
set1.clear()
print(set1)          #set( )
```

del: - The **del** keyword will delete the set completely.

<u>Example</u>

```
set1 = {“apple”, “banana”, “cherry”}
del set1
print(set1)
#This will raise an error because the set no longer exists.
```

Join Sets

There are several ways to join two or more sets in Python.

- The **union()** and **update()** methods joins all items from both sets.
- The **intersection()** method keeps ONLY the duplicates.
- The **difference()** method keeps the items from the first set that are not in the other set(s).
- The **symmetric_difference()** method keeps all items EXCEPT the duplicates.

UNION

The union() method returns a new set with all items from both sets.

Example

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = {"John", "Elena"}
set4 = {"apple", "banana", "cherry"}
myset = set1.union(set2, set3, set4)
print(myset)
#{1, 2, 3, ‘Elena’, ‘banana’, ‘a’, ‘apple’, ‘c’, ‘b’, ‘cherry’, ‘John’}
```

You can also use the **|** operator instead of the union() method, and you will get the same result.

Example

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = {"John", "Elena"}
set4 = {"apple", "banana", "cherry"}
myset = set1 | set2 | set3 | set4
print(myset)
#{1, 2, 3, ‘Elena’, ‘banana’, ‘a’, ‘apple’, ‘c’, ‘b’, ‘cherry’, ‘John’}
```

UPDATE

- The **update()** method inserts all items from one set into another.
- The update() changes the original set, and does not return a new set.

Example

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3, "a", "b", "c"}
set1.update(set2)
print(set1)
#{1, 2, 3, ‘c’, ‘a’, ‘b’}
```

INTERSECTION

- The intersection() method will return a new set, that only contains the items that are present in both sets.

Example

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3, "a", "b", "c"}
set3 = set1.intersection(set2)
print(set3)
#{‘a’, ‘b’, ‘c’}
```

- You can use the **&** operator instead of the intersection() method, and you will get the same result.

Example

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3, "a", "b", "c"}
set3 = set1 & set2
print(set3)
#{‘a’, ‘b’, ‘c’}
```

Note: - The & operator only allows you to join sets with sets, and not with other data types like you can with the intersection() method.

DIFFERENCE

- The **difference()** method will return a new set that will contain only the items from the first set that are not present in the other set.

Example

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1.difference(set2)
print(set3)
#{‘banana’, ‘cherry’}
```

- You can use the **–** operator instead of the difference() method, and you will get the same result.

Example

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 - set2
print(set3)
#{‘banana’, ‘cherry’}
```

SYMMETRIC DIFFERENCES

- The **symmetric_difference()** method will keep only the elements that are NOT present in both sets.

Example

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1.symmetric_difference(set2)
print(set3)
#{‘google’, ‘banana’, ‘cherry’, ‘microsoft’}
```

- You can use the **^** operator instead of the symmetric_difference() method, and you will get the same result.

Example

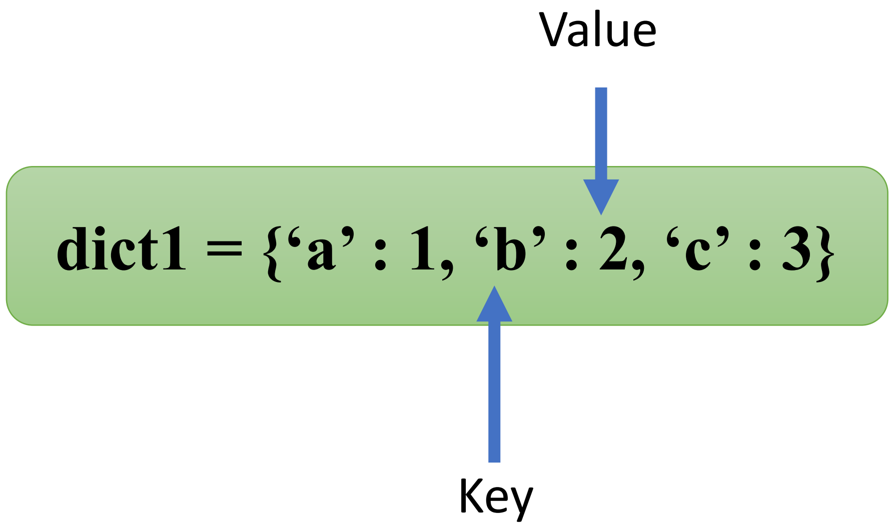
```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 ^ set2
print(set3)
#{‘google’, ‘banana’, ‘Microsoft’, ‘cherry’}
```

PYTHON DICTIONARIES

Definition: -

A dictionary can be defined as a collection of objects, values or items of different types stored in key-value pair format. These multiple key-value pairs created are enclosed within the curly braces **{ }**, and each key is separated from its value by the colon (**:**).

Example



```
dict1 = {‘a’ : 1, ‘b’ : 2, ‘c’ : 3}
```

Characteristics Of Dictionaries

- The dictionaries are **ordered**. It means that the items have a defined order, and that order will not change.
- Elements of the dictionaries cannot be accessed by index.
- Dictionaries can store various types of elements.
- Dictionaries are **mutable** or **changeable**. It means that we can change, add or remove items after the dictionary has been created.
- Dictionaries do not allow duplicate elements. Meaning that dictionaries cannot have two items with the same key.

DICTIONARY LENGTH: - To determine how many items a dictionary has, use the **len()** function.

<u>Example</u>

```
thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964,  "year": 2020}
print(len(thisdict))
```

type() – Dictionaries are defined as objects with the data type ‘dict’.

<u>Example</u>

```
thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964}
print(type(thisdict))          #<class ‘dict’>
```

Access Dictionary Items

You can access the items of a dictionary by referring to its key name, inside square brackets.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
x = dict1[“year”]
print(x)              #1964
```

There is also a method called **get()** that will give you the same result.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
x = dict1.get(“year”)
print(x)              #1964
```

GET KEYS: - The **keys()** method will return a list of all keys in the dictionary.
GET VALUES: -The **values()** method will return a list of all the values in the dictionary.
GET ITEMS: - The **items()** method will return each item in a dictionary, as tuples in a list.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
x = dict1.keys()
y = dict1.values()
z = dict1.items()
print(x)        #dict_keys([‘brand’, ‘model’, ‘year’])
print(y)        #dict_values([‘Ford’, ‘Mustang’, 1964])
print(z)
#dict_items([(‘brand’, ‘Ford’), (‘model’, ‘Mustang’), (‘year’, 1964)])
```

Change And Add Dictionary Items

You can change the value of a specific item by referring to its key name.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
dict1[“year”] = 2024
print(dict1)
#{‘brand’: ‘Ford’, ‘model’: ‘Mustang’, ‘year’: 2024}
```

Adding an item to the dictionary is done by using a new index key and assigning a value to it.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
dict1[“color”] = “red”
print(dict1)
#{‘brand’: ‘Ford’, ‘model’: ‘Mustang’, ‘year’: 1964, ‘color’: ‘red’}
```

UPDATE DICTIONARY

- The **update()** method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with the key: value pairs.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
dict1.update({“year”: 2024})
print(dict1)
#{‘brand’: ‘Ford’, ‘model’: ‘Mustang’, ‘year’: 2024}
```

- The **update()** method will update the dictionary with the items from a given argument. If a item does not exist, the item will be added.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
dict1.update({“color”: “red”})
print(dict1)
#{‘brand’: ‘Ford’, ‘model’: ‘Mustang’, ‘year’: 1964, ‘color’: ‘red’}
```

Remove Dictionary Items

There are several methods to remove items from a dictionary.

- The **pop()** method removes the item with the specified key name
- The **popitem()** method removes the last inserted item.
- The **del** keyword removes the item with the specified key name.
- The **clear()** method empties the dictionary.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964, “color”:“red”}
dict1.pop(“model”)
print(dict1)        #{‘brand’: ‘Ford’, ‘year’: 1964, ‘color’: ‘red’}
dict1.popitem()
print(dict1)        #{‘brand’: ‘Ford’, ‘year’: 1964}
del dict1[“brand”]
print(dict1)        #{‘year’: 1964}
dict1.clear()
print(dict1)        #{}
del dict1
print(dict1)        #NameError: name ‘dict1’ is not defined
```

Nested Dictionaries

A dictionary can contain dictionaries; this is called nested dictionaries.

Example

```
myfamily = {“child1” : {“name” : “John”, “year” : 2004},
            “child2” : {“name” : “Ram”, “year” : 2007},
            “child3” : {“name” : “Raju”, “year” : 2011}}
print(myfamily)
#{‘child1’: {‘name’: ‘John’, ‘year’: 2004}, ‘child2’: {‘name’: ‘Ram’, ‘year’:2007}, ‘child3’: {‘name’: ‘Raju’,
‘year’: 2011}}
```

Copy Dictionaries

Make a copy of a dictionary with the **copy()** method.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
dict2 = dict1.copy()
print(dict2)
#{“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
```

Another way to make a copy is to use the built-in function **dict()**.

Example

```
dict1 = {“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
dict2 = dict(dict1)
print(dict2)
#{“brand”: “Ford”, “model”: “Mustang”, “year”:1964}
```

Python Booleans

Booleans represent one of two values: **True** or **False**.
In programming you often need to know if an expression is True or False.
When you compare two values, the expression is evaluated and Python returns the Boolean answer.

Example:

```
    print(10 > 9)          #True
    print(10 = = 9)        #False
    print( 7 < 5)          #False
    print(15 = = 15)       #True
```

Evaluate Values and Variables

- The **bool()** function allows you to evaluate any value, and give you True or False in return.

Example

Evaluate a string and a number

```
print(bool("Hello"))          #True
print(bool(48))          #True
```

Evaluate two variables

```
x = "Hello"
y = 46
print(bool(x))          #True
print(bool(y))          #True
```

Python has a built-in function that return a Boolean value, like the **isinstance()** function, which can be used to determine if an object is of a certain data type.

Example: Check if an object is an integer or not.

```
x = 679
y = “Hello world”
print(isinstance(x, int))          #True
print(isinstance(y, int))          #False
```

Python Operators

- In Python programming, Operators in general are used to perform operations on values and variables.
- These are standard symbols used for logical and arithmetic operations.
- **OPERATORS:** These are the special symbols. E.g.- + , * , /, etc.
- **OPERAND:** It is the value on which the operator is applied.

Types Of Operators

- Arithmetic Operators → +, -, *, /, %, **, //
- Comparison Operators → <, >, <=, >=, !=, ==
- Assignment Operators → =, +=, -=, *=, %=, **=, //=
- Logical Operators → and, or, not
- Bitwise Operators → &, |, >>, <<, ~,^
- Identity Operators → is, is not
- Membership Operators → in, not in

Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations.
- Operators are +, -, *, /, %, **,//

Example

```
x = 8
y = 4
print(x + y)        #12
print(x – y)        #4
print(x * y)        #32
print(x / y)        #2.0
print(x % y)        #0
print(x ** y)       #4096
print(x // y)       #2
```

Comparison Operators

- Comparison operators are used to compare two values.
- Operators are ==, !=, >, <, >=, <=

Example

```
x = 5
y = 13
print(x == y)                 #False
print(x != y)                 #True
print(x > y)                  #False
print(x < y)                  #True
print(x >= y)                 #False
print(x <= y)                 #True
```

Assignment Operator

- Assignment operators are used to assign values to variables.
- Operators are =, +=, -=, *=, /=, %= ,**=, //=

```
x = 10
x += 2       x = x+2
print(x)     #12
x -= 2       x = x-2
print(x)     #10
x *= 2       x = x*2
print(x)     #20
x /= 2       x= x/2
print(x)     #10.0
```

```
    x //= 2        x = x//2
print(x)   #5
    x %= 3
print(x)   #2.0
    x **= 3
print(x)   #8.0
```

Logical Operators

- Logical operators are used to combine conditional statements.
- Operators are: and, or, not.

Example

```
x = 6
print(x > 1 and x < 9)              #True
print(x > 10 and x < 7)             #False
print(x > 2 or x < 5)                   #True
print(x > 7 or x < 1)                   #False
print(not(x > 3 and x < 10))            #False
print(not(x > 8 or x == 5))         #True
```

Bitwise Operator

- Bitwise operators are used to compare binary numbers that is 0 and 1.
- Operators are &, |, ^, ~, <<, >>

```
    print (5 & 3)              #1
```

The & operator compares each bit and set it to 1 if both are 1, otherwise it is set to 0

```
5 = 0000000000000101
3 = 0000000000000011
-----------------------------
1 = 0000000000000001
```

```
    print(5 | 3)               #7
```

The | operator compares each bit and set it to 1 if one or both is 1, otherwise it is set to 0

```
5 = 0000000000000101
3 = 0000000000000011
-----------------------------
7 = 0000000000000111
```

```
print(5 ^ 3)
```

The XOR(^) operator compares each bit and set it to 1 if only one bit is 1, otherwise if both are one or both are zero it is set to 0.

```
5 = 0000000000000101
3 = 0000000000000011
-----------------------------
6 = 0000000000000110
```

```
print(~1)       #-2
```

The NOT(~) operator inverts each bit that is 0 becomes 1 and 1 becomes 0.

```
 1 = 0000000000000001
------------------------------
-2  = 1111111111111110
```

```
    print(5 << 3)        #40
```

The left shift (<<) operator inserts the specified number of 0’s in this case 3 from the right and let the same amount of leftmost bits to fall off.

```
5 = 0000000000000101
-----------------------------
40= 0000000000101000
```

```
    print(9>> 3)         #1
```

The right shift (>>) operator moves each bit the specified number of times to the right. Empty holes at the left are filled with 0’s.

```
9 = 0000000000001001
-----------------------------
1 = 0000000000000001
```

Identity Operator

- The identity operator is used to check if two variables refer to the same object in memory.
- There are 2 identity operators. They are:

  is: Returns True if both variables refer to the same object in memory.

  is not: Returns True if both variables do not refer to the same object in memory.

Example

```
x = [“apple”, “banana”]
y = [“apple”, “banana”]
z = x
print(x is z)         #True
print(x is y)         #False
print(x == y)         #True
print(x is not z)     #False
print(x is not y)     #True
print(x != y)         #False
```

Membership Operators

- Membership operators are used to test if a sequence is presented in an object.
- Operators are: in, not in

Example

```
x = [“apple”, “banana”]
print(“banana” in x)          #True
print(“grapes” in x)                #False
print(“apple” not in x)       #False
print(“grapes” not in x)            #True
```

Operator Precedence

- Operator precedence describes the order in which operations are performed.
- Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first.
- Multiplication * has higher precedence than addition +, therefore multiplications are evaluated before additions.

Operators	Description
( )	Parentheses
**	Exponentiation
+x, -x, ~x	Unary plus, unary minus, and bitwise NOT
*, /, //, %	Multiplication, division, floor division, and modulus
+, -	Addition and subtraction
<<, >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <= is, is not in, not in	Comparisons, identity and membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Example

```
    print((5+3) – (7+9))                      #-8
    print(100 * (5+3) + 9 – (10 / 2))         #804.0
    print(7 + 8 – 10 * 3 / 5)                 #9.0
```

Conditional Statements

- Conditional statements, also known as decision-making statements, allow you to execute certain pieces of code based on whether a specified condition or set of conditions is true or false.

'if' Statement: Executes a block of code if a specified condition is true.

Syntax:

```
if condition:
    # code to execute if condition is true
```

Example:

```
x = 100
if x > 50:
    print("x is greater than 50")          #x is greater than 50
```

‘else’ Statement

The if-else statement allows you to execute one block of code if the condition is true, and another block of code if the condition is false.

Syntax:

```
if condition1:
    # code to execute if condition1 is true
else:
    # code to execute if condition1 is false
```

Example

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")        #x is not greater than 5
```

‘if-elif-else’ Statement

The if-elif-else statement is used to check multiple conditions. It allows you to execute different code blocks based on which condition is true.

Syntax:

```
if condition1:
    # code to execute if condition1 is true
elif condition2:
    # code to execute if condition1 is false and condition2 is true
else:
    # code to execute if condition1 and condition2 are both false
```

Example

```
x = 7
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but less than or equal to 10")
else:
    print("x is 5 or less")
#x is greater than 5 but less than or equal to 10
```

Python Loops

Python has two primitive loop commands.

1. For loop
2. While loop

For Loop

- A for loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, or string) or any iterable object.
- It executes a block of code repeatedly for each item in the sequence.

Syntax

```
for item in sequence:
        #code block to be executed
```

In this syntax,
‘item’: A variable that represents each item in the sequence during each iteration.
‘sequence’: the sequence of items over which the loop iterates.

Examples

```
    a = [‘a’, ‘b’, ‘c’]
    for i in a:
        print(i)            #a   b   c

    for x in range(3):
        print(x)            #0   1   2

    for x in “Mangalore”
        print(x)            #M      a      n      g      a      l      o      r      e
```

To calculate the sum

```
    numbers = [2, 4, 6, 8, 10]
    sum = 0
    for number in numbers:
        sum += number
    print(“The total sum is: ”, sum) #The total sum is: 30
```

The break Statement: - With the break statement we can stop the loop before it has looped through all the items.

Example 1

```
f = [‘a’, ‘b’, ‘c’]
for x in f:
    print(x)
    if x==‘b’:
        break
```

Output

```
a
b
```

Example 2

```
f = [‘a’, ‘b’, ‘c’]
for x in f:
    if x==‘b’:
        break
    print(x)            #a
```

The continue statement: - With the continue statement we can stop the current iteration of the loop, and continue with the next.

Example

```
    f = [‘a’, ‘b’, ‘c’]
    for x in f:
        if x==‘b’:
            continue
        print(x)            #a   c
```

Nested Loops: - A nested loop is a loop inside a loop. The ‘inner loop’ will be executed one full time for each iteration of the ‘outer loop’.

Example:

```
colors = [“red”, “yellow”]
fruits = [“apple”, “banana”]
for color in colors:
    for fruit in fruits:
        print(color, fruit)
```

Output

```
red apple
red banana
yellow apple
yellow banana
```

The pass statement: - for loops cannot be empty, but if for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

Example

```
for x in range(5):
    pass
```

While Loop

- With the while loop we can execute a set of statements as long as a condition is true.

Syntax

```
while condition:
    #code block to be executed
```

Example

```
x = 0
while x < 5:
    print(x)
    x += 1
```

Output

```
0    1    2    3    4
```

The break Statement: - With the break statement we can stop the loop even if the while condition is true.

Example

```
i = 0
while i < 5:
    print(i)
    if i==2:
        break
    i+=1
```

The continue statement: - With the continue statement we can stop the current iteration, and continue with the next iteration.

Example

```
x = 1
while x < 6:
    x += 1
    if x ==3:
        continue
    print(x)
```

Functions

- A function is a block of code which only runs when it is called.
- You can pass data, know as parameters or arguments, into a function.
- A function can return data as a result.

Creating a Function: - In Python a function is defined using the **def** keyword.

Example

```
def my_function():
    print(“Hello from a function”)
```

Calling a Function: - To call a function, use the function name followed by parenthesis.

Example

```
def my_function():
    print(“Hello from a function”)
my_function()
```

Output

```
Hello from a function
```

Arguments: - In Python, arguments refer to the values that are passed to a function when it is called. Functions can accept zero or more arguments, which are used as inputs to perform some computation or task within the function.

Syntax

```
def function_name(arg1, arg2,……., argN):
    #Function body
```

Example

```
def my_age(age):
    print("I am" , age)
my_age(20)
my_age(25)
```

Output

```
I am 20
I am 25
```

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example: This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
    print(fname + " " + lname)
my_function("Emil", "Refsnes")        #Emil Refsnes
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

Example: If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")        #The youngest child is Linus
```

Keyword Arguments
You can also send arguments with the key = value syntax. This way the order of the arguments does not matter.
Example:
```
    def my_function(child3, child2, child1):
        print("The youngest child is " + child3)
    my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```
Output: - The youngest child is Linus

Arbitrary Keyword Arguments, **kwargs
If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.
This way the function will receive a *dictionary* of arguments, and can access the items accordingly:
Example: If the number of keyword arguments is unknown, add a double ** before the parameter name:
```
    def my_function(**kid):
        print("His last name is " + kid["lname"])
    my_function(fname = "Tobias", lname = "Refsnes")
```
Output: - His last name is Refsnes

Default Parameter Value

If we call the function without argument, it uses the default value:

Example:

```
def my_function(country = "Norway"):
    print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Output:

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example:

```
def my_function(food):
    for x in food:
        print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)     #apple  banana   cherry
```

Return Values

To let a function return a value, use the **return** statement.

Example

```
def my_function(x):
    return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))        #15      25   45
```

Positional-Only Arguments

You can specify that a function can have ONLY positional arguments, or ONLY keyword arguments.
To specify that a function can have only positional arguments, add **, /** after the arguments:

Example 1

```
def my_function(x, /):
    print(x)
my_function(3)        #3
```

Without the , / you are actually allowed to use keyword arguments even if the function expects positional arguments:

Example 2

```
def my_function(x):
    print(x)
my_function(x = 3)       #3
```

Note: But when adding the , / you will get an error if you try to send a keyword argument(ex: x=3)

Error example:

```
def my_function(x, /):
    print(x)
my_function(x = 3)
```

Keyword-Only Arguments

To specify that a function can have only keyword arguments, add ***,** *before* the arguments:

Example:

```
def my_function(*, x):
    print(x)
my_function(x = 3)       #3
```

Without the *, you are allowed to use positionale arguments even if the function expects keyword arguments:

Example:

```
def my_function(x):
    print(x)
my_function(3)       #Output: 3
```

Note: But when adding the *, / you will get an error if you try to send a positional argument:

Error Example:

```
def my_function(*, x):
    print(x)
my_function(3)
```

Combine Positional-Only and Keyword-Only

You can combine the two argument types in the same function.

Any argument *before* the / , are positional-only, and any argument *after* the *, are keyword-only.

Example:

```
def my_function(a, b, /, *, c, d):
    print(a + b + c + d)
my_function(5, 6, c = 7, d = 8)      #26
```

Lambda function

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax: lambda arguments : expression

Example

```
x = lambda a, b, c, d :a +b-c*d
print(x(5))

y = lambda a, b : a * b
print(y(5, 6))
```

Global and Local Variables:

Variables that are created outside of a function are known as global variables.
Global variables can be used by everyone, both inside of functions and outside.

Example:

```
x = "awesome"
def myfunc():
        print("Python is " + x)
myfunc()                                #Python is awesome
```

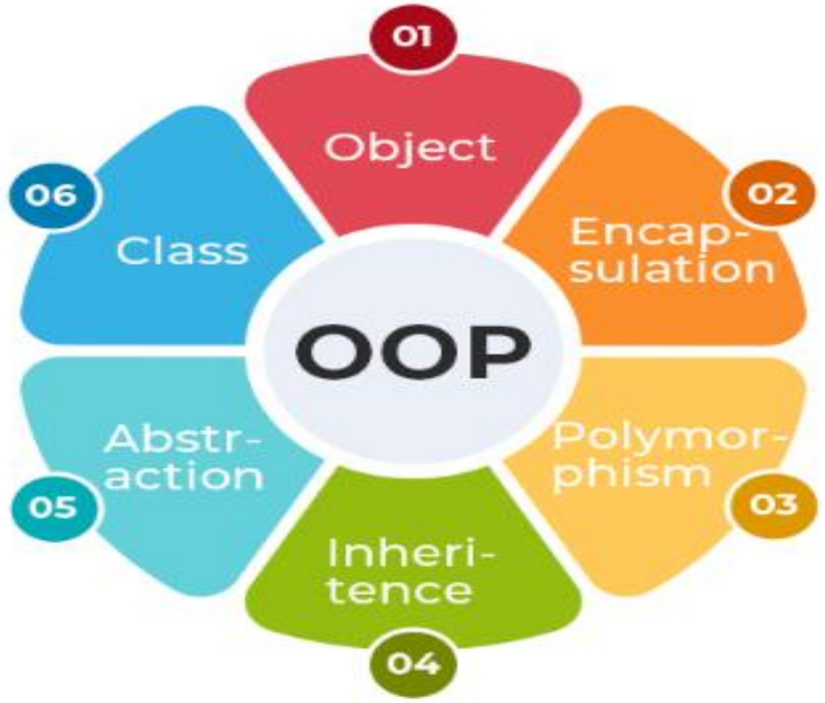
If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as global and with the original value.

Example:

```
x = “awesome”
def myfunc():
        x = “fantastic”
        print(“Python is “ + x)              #Python is fantastic
myfunc()
print(“Python is “ + x)                      #Python is awesome
```

PYTHON OOPs CONCEPT

- Object Oriented Programming is a fundamental concept in Python, empowering developers to build modular, maintainable, and scalable applications.
- By understanding the core OOP principles (classes, objects, inheritance, encapsulation, polymorphism, and abstraction), programmers can leverage the full potential of Python OOP capabilities to design elegant and efficient solutions to complex problems.
- OOPs is a way of organizing code that uses objects and classes to represent real-world entities and their behavior.
- In OOPs, object has attributes that has specific data and can perform certain actions using methods.


OOP
01
Object
02
Encap-
sulation
03
Polymor-
phism
04
Inheri-
tence
05
Abstr-
action
06
Class

PYTHON CLASS

- A class is a collection of objects.
- Classes are blueprints for creating objects.
- A class defines a set of attributes and methods that the created objects (instances) can have.

Some points on Python class:

- Classes are created by keyword **class**.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Example: Myclass.Myattribute

CREATING A CLASS

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute
```

PYTHON OBJECTS

- An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

An object consists of:

- **State:** It is represented by the attributes and reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object and reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Creating Object

- Creating an object in Python involves instantiating a class to create a new instance of that class.
- This process is also referred to as object instantiation.

Example

```
class Dog:
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute

# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.name)
Print(dog1.age)
```

SELF PARAMETER

- self parameter is a reference to the current instance of the class. It allows us to access the attributes and methods of the object.

Example

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute

dog1 = Dog("Buddy", 3)  # Create an instance of Dog
dog2 = Dog("Charlie", 5)  # Create another instance of Dog

print(dog1.name, dog1.age, dog1.species)  # Access instance and class attributes
print(dog2.name, dog2.age, dog2.species)  # Access instance and class attributes
print(Dog.species)  # Access class attribute directly
```

__init__ METHOD

- __init__ method is the constructor in Python, automatically called when a new object is created.
- It initializes the attributes of the class.

Example

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

dog1 = Dog("Buddy", 3)
print(dog1.name)
```

PYTHON INHERITANCE

- Inheritance allows a class (child class) to acquire properties and methods of another class (parent class).
- It supports hierarchical classification and promotes code reuse.

Types of Inheritance:

1. **Single Inheritance:** A child class inherits from a single parent class.
2. **Multiple Inheritance:** A child class inherits from more than one parent class.
3. **Multilevel Inheritance:** A child class inherits from a parent class, which in turn inherits from another class.

```
# Single Inheritance
class Dog:
    def __init__(self, name):
        self.name = name

    def display_name(self):
        print(f"Dog's Name: {self.name}")

class Labrador(Dog):  # Single Inheritance
    def sound(self):
        print("Labrador woofs")

# Multilevel Inheritance
class GuideDog(Labrador):  # Multilevel Inheritance
    def guide(self):
        print(f"{self.name} Guides the way!")

# Multiple Inheritance
class Friendly:
    def greet(self):
        print("Friendly!")
```

```
class GoldenRetriever(Dog, Friendly):
        def sound(self):
                print("Golden Retriever Barks")

# Example Usage
lab = Labrador("Buddy")
lab.display_name()
lab.sound()

guide_dog = GuideDog("Max")
Guide =GuideDog(“Don”)
guide_dog.display_name()
guide_dog.guide()

retriever = GoldenRetriever("Charlie")
retriever.display_name()
retriever.greet()
retriever.sound()
```

PYTHON POLYMORPHISM

- Polymorphism allows methods to have the same name but behave differently based on the object’s context.
- It can be achieved through method overriding or overloading.

Types of Polymorphism

1.Compile-Time Polymorphism: This type of polymorphism is determined during the compilation of the program. It allows methods or operators with the same name to behave differently based on their input parameters or usage. It is commonly referred to as method or operator overloading.

2.Run-Time Polymorphism: This type of polymorphism is determined during the execution of the program. It occurs when a subclass provides a specific implementation for a method already defined in its parent class, commonly known as method overriding.

```
# Parent Class
class Dog:
    def sound(self):
        print("dog sound")  # Default implementation

# Run-Time Polymorphism: Method Overriding
class Labrador(Dog):
    def sound(self):
        print("Labrador woofs")  # Overriding parent method

class Beagle(Dog):
# Overriding parent method

# Run-Time Polymorphism
dogs = [Dog(), Labrador(), Beagle()]
for dog in dogs:
    dog.sound()  # Calls the appropriate method based on the
object type
```

```
# Compile-Time Polymorphism: Method
Overloading Mimic
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c  # Supports multiple ways
to call add()

# Compile-Time Polymorphism (Mimicked using
default arguments)
calc = Calculator()
print(calc.add(5, 10))  # Two arguments
print(calc.add(5, 10, 15))  # Three arguments
```

PYTHON ENCAPSULATION

- Encapsulation is the bundling of data (attributes) and methods (functions) within a class, restricting access to some components to control interactions.
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name  # Public attribute
        self._breed = breed  # Protected attribute
        self.__age = age  # Private attribute

    # Public method
    def get_info(self):
        return f"Name: {self.name}, Breed: {self._breed},
Age: {self.__age}"

    # Getter and Setter for private attribute
    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Invalid age!")
```

```
# Example Usage
dog = Dog("Buddy", "Labrador", 3)

# Accessing public member
print(dog.name)  # Accessible

# Accessing protected member
print(dog._breed)  # Accessible but
discouraged outside the class

# Accessing private member using getter
print(dog.get_age())

# Modifying private member using setter
dog.set_age(5)
print(dog.get_info())
```

DATA ABSTRACTION

- Abstraction hides the internal implementation details while exposing only the necessary functionality.
- It helps focus on “what to do” rather than “how to do it.”

Types of Abstraction:

•**Partial Abstraction:** Abstract class contains both abstract and concrete methods.

•**Full Abstraction:** Abstract class contains only abstract methods (like interfaces).

```
class Dog():  # Abstract Class
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def sound(self):  # Abstract Method
        pass

    def display_name(self):  # Concrete Method
        print(f"Dog's Name: {self.name}")

class Labrador(Dog):
    def sound(self):
        print("Labrador Woof!")

class Beagle(Dog):
    def sound(self):
        print("Beagle Bark!")
```

```
# Example Usage
dogs = [Labrador("Buddy"), Beagle("Charlie")]
for dog in dogs:
    dog.display_name()  # Calls concrete method
    dog.sound()  # Calls implemented abstract
method
```

PYTHON EXCEPTION HANDLING

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The else block lets you execute code when there is no error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

Example: The try block will generate an exception, because x is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")         #An exception occurred
```

Many Exceptions:

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error.

Example:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")          #Variable x is not defined
```

Else:

You can use the else keyword to define a block of code to be executed if no errors were raised:

Example: In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")          #Hello    Nothing went wrong
```

Finally:

The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example:

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")        #Something went wrong       The 'try except' is finished
```

Raise an exception

- As a Python developer you can choose to throw an exception if a condition occurs.
- To throw (or raise) an exception, use the raise keyword.
- The raise keyword is used to raise an exception.
- You can define what kind of error to raise, and the text to print to the user.

Example 1: Raise an error and stop the program if x is lower than 0:

```
x = -1
if x < 0:
    raise Exception("Sorry, no numbers below zero")
#Exception: Sorry, no number below zero
```

Example 2: Raise a TypeError if x is not an integer:

```
x = "hello"
if not type(x) is int:
    raise TypeError("Only integers are allowed")
# TypeError: Only integers are allowed
```

Python File Handling

- Python file handling is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.

Python File Open

- The key function for working with files in Python is the **open()** function.
- The open() function takes two parameters: **filename** and **mode.**

There are four different modes for opening a file:

- **‘r’ – Read –** Default value. Opens a file for reading, error if the file does not exist.
- **‘a’ – Append –** Opens a file for appending, creates the file if it does not exist.
- **‘w’ – Write –** Opens a file for writing, creates the file if it does not exist.
- **‘x’ – Create –** Creates the specified file, returns an error if the file exists.

In addition you can specify if the file should be handled as binary or text mode.

- **‘t’ – Text –** Default value. Text mode.
- **‘b’ – Binary –** Binary mode.

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open(“demofile.txt”)
```

The code above is same as:

```
f = open(“demofile.txt”, “rt”)
```

Because “r” for read, and “t” for text are the default values, you do not need to specify them.

Note: - Make sure the file exists, or else you will get an error.

Example of File

demofile.txt is a file and its content are: -

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

Read a File: -

Assume we have the demofile.txt file, located in the same folder as Python.

To open the file, use the built-in **open()** function.

The **open()** function returns a file object, which has a **read()** method for reading the content of the file.

```
f = open(“demofile.txt”, “r”)
print(f.read())
```

Output: -

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

If the file is located in a different location, you will have to specify the file path, like this:

```
f = open(“D:\\myfiles\\demofile.txt”, “r”)
print(f.read())
```

Read only Parts of the File: -

By default the **read()** method returns the whole text, but you can also specify how many characters you want to return.

Example Return the first 10 characters of the file:

```
f = open(“demofile.txt”, “r”)
print(f.read(10))          #Hello! Wel
```

Read Lines: -

You can return one line by using the **readline()** method:

Example

```
f = open(“demofile.txt”, “r”)
print(f.readline())        #Hello! Welcome to demofile.txt
```

By calling **readline()** two times, you can read the first two lines.

Example

```
f = open(“demofile.txt”, “r”)
print(f.readline())        #Hello! Welcome to demofile.txt
print(f.readline())        #This file is for testing purposes.
```

Close Files: -

It is a good practice to always close the file when you are done with it.

Example

```
f = open(“demofile.txt”, “r”)
print(f.readline())            #Hello! Welcome to demofile.txt
f.close()
```

Python File Write:

Write to an Existing File: -

To write to an existing file, you must add a parameter to the **open()** function.

‘a’ – Append – will append to the end of the file.

‘w’ – Write – will overwrite any existing content.

Example

```
f = open(“demofile.txt”, “a”)
f.write(“Now the file has more content!”)
f.close()

f = open(“demofile.txt”, “r”)
print(f.read())
```

Output:

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
Now the file has more content!
```

To overwrite the content in the existing file use “w” mode.

Example

```
f = open(“demofile.txt”, “w”)
f.write(“Woops! I have deleted the content!”)
f.close()

f = open(“demofile.txt”, “r”)
print(f.read())
```

Output:

```
Woops! I have deleted the content!
```

Note: The “w” method will overwrite the entire file.

Create a new file: -

```
f = open(“myfile.txt”, “x”)
```

Result: A new empty file is created.

You can also create a new file using “w” mode if it does not exist.

```
f = open(“mynewfile.txt”,”w”)
```

Python Delete File: -

To delete a file, you must import the OS module, and run its **os.remove()** function.

Example:

```
import os
os.remove(“demofile.txt”)
```

Check if File Exist

To avoid getting an error, you might want to check if the file exists before you try to delete it.

Example:

```
import os
if os.path.exists(“demofile.txt”):
	os.remove(“demofile.txt”)
else:
	print(“The file does not exist”)
```