WELCOME TO PYTHON CLASS

PYTHON HISTORY & FEATURES

PYTHON HISTORY & FEATURES

General-Purpose:

Python is a versatile programming language designed to be used for a wide range of applications and domains.

Created by Guido van Rossum:

Python was created by Guido van Rossum and development began in December 1989, with the first release in February 1991.

Interpreted:

Python is an interpreted language, which means it doesn't require a separate compilation step and can execute code line-by-line.

Interactive:

Python allows interactive development, where users can execute code and see results immediately using the Python interpreter.

Object-Oriented:

Python supports object-oriented programming paradigms, allowing developers to create and use classes and objects.

High-Level:

Python is a high-level language that abstracts away low-level details, making reading and writing code easier.

Dynamically-Typed:

Python is dynamically-typed, meaning variable types are determined at runtime, and users don't need to specify them explicitly.

Garbage-Collected:

Python has automatic memory management through garbage collection, which frees up memory occupied by unused objects.

Popularity:

Python has gained immense popularity due to its simplicity, readability, and community support.

Versatility:

Python is used for web development, data analysis, machine learning, automation, scripting, and more, making it a powerful and flexible language.

Licensing:

Python's source code is available under the Python Software Foundation License (PSFL), an open-source license similar to the BSD license

KEYWORDS

Python Keywords:

Keywords in Python are reserved words that have specific meanings and purposes in the language. These words cannot be used as variable names or identifiers because they are already predefined with their specific functionalities.

```
import keyword

print(keyword.kwlist)
```

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

Indentation in Python

- In most programming languages, indentation is used primarily for readability and has no effect on the code's functionality.
- However, in Python, indentation has a significant role in the language's syntax and semantics.
- In Python, indentation is used to define blocks of code, such as loops, conditional statements, function definitions, and classes., Unlike other languages that use braces or keywords to denote blocks,
- Python uses indentation to specify the beginning and end of a block.

Example of Indentation:-

```
if condition:
    # Code block executed if the condition is true
    statement1
    statement2
else:
    # Code block executed if the condition is false
    statement3
    statement4
```

Command line arguments.

- Comments in Python are the lines in the code that are ignored by the interpreter during the execution of the program.
- Comments enhance the readability of the code and help the programmers to understand the code very carefully.
- There are three types of comments in Python.

Single line comments

```
# Python program to demonstrate comments
# sample comment

name = "hello world!"
print(name)#print a comment line
```

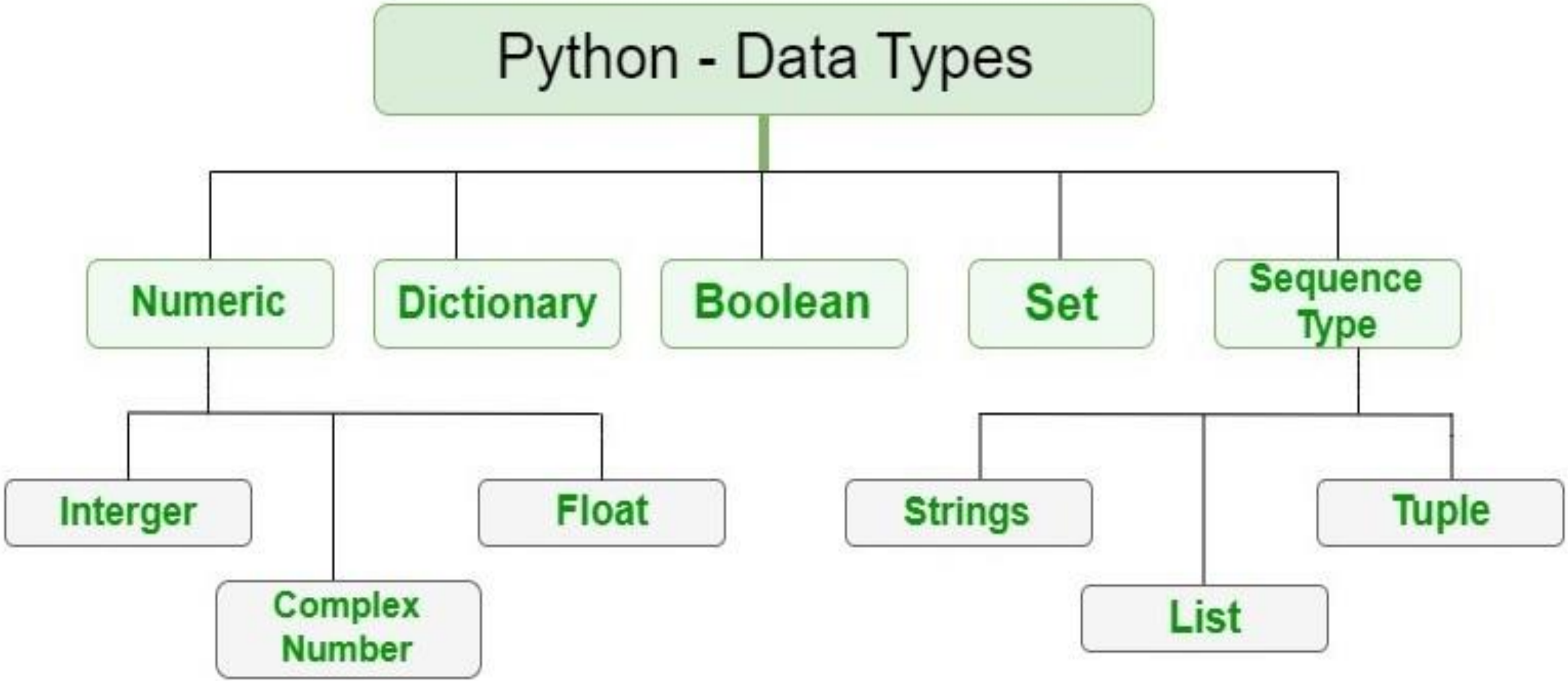
Multiline Comments

```
""" python program to demonstrate
multiline comments"""

print("multiline comments")
```

Docstring Comments

```
def multiply(a, b):
        """ Multiplies the value of a and b"""
        return a*b
```

```
# Print the docstring of the multiply function

print(multiply.__doc__)
```


Python - Data Types
Numeric
Dictionary
Boolean
Set
Sequence Type
Interger
Complex Number
Float
Strings
List
Tuple

INTEGER – int()

- Integer is a fundamental data type in Python.
- It represents whole numbers without any fractional or decimal component.
- Integers can be positive, negative, or zero.
- They are commonly used for counting, indexing, and performing arithmetic operations.
- Integers are declared using plain digits without any decimal points or quotes.
- Examples of integers in Python:
  - Positive integer: x = 10
  - Negative integer: y = -5
  - Zero: z = 0

STRING – str() & chr()

- String is a data type in Python used to represent a sequence of characters.
- Characters within a string can be letters, numbers, symbols, or spaces.
- Strings are immutable, meaning their values cannot be changed after creation. However, you can create new strings based on existing ones.
- Strings are enclosed in either single quotes (' ') or double quotes (" "). Both forms are valid and can be used interchangeably.

STRING

- Examples of strings in Python:
  - Using single quotes: 'Hello, World!'
  - Using double quotes: "Python is fun!"
- You can perform various operations on strings, such as concatenation (joining), slicing (extracting substrings), and repetition.
- Python provides a wide range of string methods for manipulating and processing strings, such as converting cases, finding substrings, and replacing text.
- Strings can be formatted using placeholders and formatting methods to insert variables or values dynamically.

STRING

- Escape sequences allow you to include special characters within strings, such as newline \n, tab \t, or backslash \\.
- String indexing starts from 0, where the first character is at index 0, the second character at index 1, and so on.
- You can use negative indices to access characters from the end of the string, with -1 representing the last character.
- To get the length of a string, you can use the built-in len() function.

Float- float()

- float is a data type that represents real numbers (numbers with decimal points).
- It stores floating-point values, which can be positive or negative and have a fractional part.

```
# Declaration and Initialization
a = 3.14
# Mathematical Operations
a = 2.5
b = 1.3
result_addition = a + b
```

Complex-complex()

- The complex data type is used to represent complex numbers.
- A complex number consists of two parts: a real part and an imaginary part, which is denoted by "j" in Python(e.g., 3 + 2j).

```
# Declaration and Initialization
 a= 3 + 2j

# Mathematical Operations
a = 2 + 3j
b = 1 - 1j
c= a + b
```

VARIABLE

- In Python a variable is a symbolic name that represents a value stored in the computer's memory.
- It acts as a container for holding data, and its value can be changed or modified during the execution of a program.
- Variables play a crucial role in programming as they allow us to store and manipulate data, making our code more dynamic and flexible.

Variable Names: must follow

- It can contain letters (a-z, A-Z), digits (0-9), and underscores (_).
- It must start with a letter or an underscore (not a digit).
- Variable names are case-sensitive, meaning age, Age, and AGE are considered different variables.
- Not a keywords
- Don’t use spaces

```
                                  # Variables with different data types
name = "Bob"                      # string
a = 40                            # integer
a= True                           # Boolean
b = [85, 90, 78]                  # list
```

Operators in python

- In Python, operators are symbols or special keywords used to perform various operations on data.

- Operators allow you to manipulate values, perform arithmetic computations, compare values, combine or modify data, and more.

- Python supports a wide range of operators, which can be categorized into different types:

Arithmetic Operators:

```
a = 10
b = 3
addition = a + b      # Addition (+)
subtraction = a - b   # Subtraction (-)
multiplication = a * b   # Multiplication (*)
division = a / b        # Division (/)
modulo = a % b          # Modulo (%)
exponentiation = a ** b  # Exponentiation (**)
floor_division = a // b  # Floor Division (//)
```

Comparison Operators:

```
a = 5
b = 8
e = a == b        # Equal to (==)
n = a != b    # Not equal to (!=)
g = a > b   # Greater than (>)
l = a < b      # Less than (<)
g = a >= b  # Greater than or equal to (>=)
l = a <= b     # Less than or equal to (<=)
```

Logical Operators:

a = True

b = False

C = a and b # Logical AND (True if both are True)

C = a or b # Logical OR (True if at least one is True)

C = not a # Logical NOT (Inverts the value)

Assignment Operators:

```
a = 10
b = 5
# Various assignment operators
a += b    # a = a + b
a -= b    # a = a - b
a *= b    # a = a * b
a /= b    # a = a / b
a %= b    # a = a % b
a **= b   # a = a ** b
a //= b   # a = a // b
```

Membership Operators:

- a= [1, 2, 3, 4]
- b= 3 in a # True if 3 is in the list
- i = 5 not in a # True if 5 is not in the list

Identity Operators:

- x = 10
- y = 10
- z = [1, 2, 3]

- **A = x is y # True if x and y refer to the same object in memory**
- **B = x is not z # True if x and z do not refer to the same object**

Strings in python

- In Python, a string is a sequence of characters enclosed within
  single quotes (' ‘),
  double quotes (" "), or
  triple quotes (""" """ or ''' '’’).
- Strings are used to represent text data and are one of the fundamental data types in Python.
- They allow you to store and manipulate text, such as names, sentences, or any sequence of characters.

- # Declaration and Initialization

a = "Alice"

b = 'Hello, how are you?’

c = “”” This is a multi-line string.

- You can use triple quotes for multi-line text."""“

Accessing Characters

message = "Hello"

print(message[0])   # Output: "H"

print(message[2])   # Output: "l"

String Concatenation

- a= “Ram"
- b= “Krishna"
- c= a + " " + b
- print(c) # Output: “Ram Krishna"

String Methods

- text = "Hello, World!"
- print(text.lower()) # Output: "hello, world!"
- print(text.upper()) # Output: "HELLO, WORLD!"
- print(text. find("World")) # Output: 7 (index where "World" starts)
- print(text.replace("Hello", "Hi")) # Output: "Hi, World!"

String Slicing

```
a = "Welcome to Python class"
b = a[8:14]   # Extracts "to Pyt"
```

Escape Characters

```
a= "This is a \"quote\" inside a string."
print(a)   # Output: "This is a "quote" inside a string."
```

Getting user input

- In Python, you can get user input from the console using the input() function.
- The input() function allows the user to enter text, and the text is returned as a string.
- You can then store the user input in a variable and use it in your program.

```
name = input("Please enter your name: ")

print("Nice to meet you , " + name + "!")
```

Typecasting: Input() always returns the user input as a stringyou need to convert it using appropriate typecasting functions like int() or float()

Ex:- a = int(input("Enter a Number: "))

print(a*2)

Download and install idle For Python:-(www.python.org)

- My first Python program:-
- Open→idle → file→ New file→ type
- print("Hello world")
- →save or press F5 (fn+F5)

Loops and Control structure

Control structure:-

- Simple if
- If-else
- If-elif-else
- nestedif

Simple if

- If the condition inside the parentheses evaluates to true, the code block within the curly braces will be executed.
- Otherwise, if the condition is false, the code block will be skipped, and the program will continue with the next instructions after the "if" statement.

```
if (condition) {
   // Code block to be executed if the condition is true
}
```

If - else

- The "if-else" statement is another fundamental control structure in programming, closely related to the "if" statement.

- It allows the program to make decisions based on conditions but provides an alternative code block to be executed when the initial condition is false.

```
if (condition) {
    // Code block to be executed if the condition is true
} else {
    // Code block to be executed if the condition is false
}
```

If – else working

1. The program first evaluates the condition inside the parentheses.

1. If the condition is true, the code block inside the first set of curly braces will be executed.

1. If the condition is false, the code block inside the second set of curly braces (after "else") will be executed.

If-elif-else

- If condition 1 is true, the code block inside the first indented block will be executed, and the rest of the "if-elif-else" statement will be skipped.
- If condition 1 is false, Python moves on to the next condition specified by elif.
- If condition 2 is true, the code block inside the second indented block (under elif) will be executed, and the rest of the "if-elif-else" statement will be skipped.
- If none of the conditions (including the initial "if" condition and all the elif conditions) are true, the code block inside the indented block under else will be executed.

syntax:-

```
if condition1:
    # Code block to be executed if condition1 is true
elif condition2:
    # Code block to be executed if condition1 is false and condition2 is true
else:
    # Code block to be executed if both condition1 and condition2 are false
```

Nested If in Python

- In programming, conditional statements are used to make decisions based on certain conditions.
- Nested if statements are a way to have one if statement inside another, allowing you to check multiple conditions in a more complex way.

Syntax:-

```
if condition1:
    # Code to execute if condition1 is True
    if condition2:
        # Code to execute if both condition1 and condition2 are True
    else:
        # Code to execute if condition1 is True but condition2 is False
else:
    # Code to execute if condition1 is False
```

Loops

While loop:-

- "while loop" is a control structure that repeatedly executes a block of code as long as a specified condition remains true.
- It allows you to create repetitive tasks without having to write the same code over and over again.

Syntax:-

```
while condition:
    # Code block to be executed repeatedly as long as the condition is true
```

1. The program first evaluates the condition inside the parentheses.
2. If the condition is true, the code block inside the indented block will be executed.
3. After the code block is executed, the program returns to the beginning of the "while loop" and checks the condition again.
4. If the condition is still true, the code block is executed again, and this process continues in a loop until the condition becomes false.
5. Once the condition becomes false, the program exits the loop, and the control moves to the next statement after the "while loop."

For loop

- "for loop" is a control structure that allows you to iterate over a sequence of elements, such as lists, strings, or ranges.

- It executes a block of code for each item in the sequence, making it useful for performing repetitive tasks on each element.

Syntax:-

```
for element in sequence:
        block
```

1.The "for loop" starts by taking the first element from the sequence and assigning it to the variable element.

2.The code block inside the indented block is then executed with the current value of the element.

3.After executing the code block, the loop moves on to the next element in the sequence, and the process repeats.

4.This continues until the loop has iterated over all elements in the sequence.

Break and Continue statements

- The “break” statement is used to exit or "break out“ of a loop prematurely.

- It is commonly used inside loops, such as for loops and while loops, to terminate the loop's execution before it reaches its natural end based on the loop condition.

Example:-

```
count = 0

while True:
    print(count)
    count += 1
    if count == 5:
        break
```

Continue Statements

- The continue statement is used to skip the rest of the current iteration of a loop and move to the next iteration.
- When the continue statement is encountered within a loop, the remaining code within that iteration is skipped, and the loop immediately starts the next iteration, without executing the code that follows the continue statement.

Example

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Special data types

In Python, special data types refer to those that have specific characteristics or behaviors beyond the basic built-in data types like integers, floats, strings.

Lists:-

- In Python, a list is a built-in data type that represents an ordered collection of items.
- Lists are versatile and widely used due to their ability to hold elements of different data types and their mutable nature, meaning you can modify the elements they contain after creation.

Creating Lists: You can create a list by enclosing a comma-separated sequence of values within square brackets ‘[ ]’.

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

Accessing Elements: List elements are accessed using zero-based indexing. You can access an element by providing its index within square brackets.

```
print(my_list[0])  # Outputs: 1
```

MODIFYING ELEMENTS: Lists are mutable, so you can change the value of an element using its index:

```
my_list[2] = 10
print(my_list)  # Outputs: [1, 2, 10, 4, 5]
```

Appending and Extending: You can add elements to the end of a list using the 'append()' method, or you can concatenate another list using the 'extend()' method:

```
my_list.append(6)
new_elements = [7, 8, 9]
my_list.extend(new_elements)
```

Slicing Lists

Slicing allows you to extract a portion of a list by specifying start and end indices:

```
sub_list = my_list[2:5]  # Returns elements from index 2 to 4 (not including 5)
```

List Methods :

Lists have numerous built-in-methods, such as ‘insert()’,’remove()’,’pop()’,and ‘index()’,which allow you to manipulate and search for elements within the list.

Length of a List: You can find the number of elements in a list using the len()

Iterating Over a List: You can use a ‘for’ loop to iterate through the elements of a list:

```
for item in my_list:
    print(item)
```

Tuples

- A tuple in Python is another built-in data type that represents an ordered, immutable collection of items.
- Unlike lists, tuples cannot be modified after creation, making them suitable for situations where you need a fixed set of values that should not change.

Creating Tuples:

Tuples are created by enclosing a comma-separated sequence of values within parentheses

```
my_tuple = (1, 2, 3, 4, 5)
```

Accessing Elements:

Similar to lists, tuple elements are accessed using zero-based indexing. You can retrieve an element by providing its index within square brackets:

```
print(my_tuple[0])  # Outputs: 1
```

Immutable Nature: Tuples are immutable, which means you cannot change the value of an element once the tuple is created:

```
# This will raise an error
my_tuple[2] = 10
```

Use Cases: Tuples are often used when you want to group related data together that shouldn't be changed. For example, coordinates (x, y) or RGB color values (red, green, blue) can be represented using tuples.

Multiple Assignment::`q

You can assign multiple values to multiple variables using tuple unpacking:

Tuple Methods: index() and count()

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

Concatenation and Repetition:

```
concatenated_tuple = (1, 2) + (3, 4)  # Results in (1, 2, 3, 4)
repeated_tuple = (0,) * 3  # Results in (0, 0, 0)
```

Sets

In Python, a set is a built-in data type that represents an unordered collection of unique elements. Sets are useful for tasks that involve checking for membership, removing duplicates from a collection, and performing set operations like union, intersection, and difference.

Creating Sets: Sets are created by enclosing a comma-separated sequence of values within curly braces '{}' or by using the ' set()' constructor

Unique Elements: Sets only store unique elements. If you try to add duplicate elements, they will be ignored:

```
my_set = {1, 2, 2, 3}  # Results in {1, 2, 3}
```

Membership Testing: You can quickly check if an element is present in a set using the 'in' keyword.

```
print(2 in my_set)  # Outputs: True
```

Set Operations: set1 = {1, 2, 3} set2 = {3, 4, 5}

union_result = set1 | set2 # {1, 2, 3, 4, 5}
intersection_result = set1 & set2 # {3}
difference_result = set1 - set2 # {1, 2}
symmetric_difference_result = set1 ^ set2 # {1, 2, 4, 5}

set1 = {1, 2, 3}
set2 = {3, 4, 5}
union(other_set):
union_set = set1.union(set2)

intersection(other_set):
intersection_set = set1.intersection(set2)
difference(other_set):
difference_set = set1.difference(set2)

Dictionaries

In Python, a dictionary is a built-in data type that represents an unordered collection of key-value pairs. Each key in the dictionary is unique and maps to a corresponding value. Dictionaries are also known as associative arrays or hash maps in other programming languages.

Creating Dictionaries: Dictionaries are created by enclosing comma-separated key-value pairs within curly braces ‘{}’ . Each key is followed by a colon ‘:’ and its corresponding value.

```
my_dict = {'name’: Ram', 'age’: 31, 'city': 'New York'}
```

Accessing Values: You can access the value associated with a specific key by using the key within square brackets ‘[]’.

```
print(my_dict['name'])  # Outputs: John
```

Modifying Values: Dictionaries are mutable, so you can change the value associated with a key after the dictionary is created:

```
my_dict['age'] = 31
```

Adding and Removing Key-Value Pairs: 'del' statement or the pop()

```
my_dict['occupation'] = 'Engineer'
del my_dict['city']
removed_value = my_dict.pop('age')
```

Dictionary Methods: Dictionaries have several built-in methods, including keys() , values() and items(), which allow you to retrieve keys, values, or key-value pairs as iterable views.

Iterating Over Dictionaries: You can use a 'for' loop to iterate through the keys, values, or key-value pairs of a dictionary:

```
for key in my_dict:
        print(key, my_dict[key])
```

FUNCTION, MODULES
&
PACKAGE IN PYTHON

Python Functions

- Functions are blocks of code in programming that perform specific tasks or operations.
- They help organize code into reusable and modular units, making it easier to manage and maintain complex programs.
- Functions play a fundamental role in many programming languages, including Python.

There are several types of functions based on their behavior and usage:-

Build-in Function

These are functions that are included in Python's standard library and are available for use without needing to import any additional modules.

Ex :- ‘print()’ , ‘len()’ , ‘range()’,etc

User-defined function:-

These are functions that you define yourself to perform specific tasks. They allow you to encapsulate a piece of code into a reusable function.

i) function call

syntax:-

```
function_name(parameter)
```

ii) function definition(def – keyword)

Syntax:-

```
def function_name(parameter):
    block
    return result
```

Anonymous Function:

Lambda functions are small, anonymous functions that can have any number of arguments, but only one expression. They are often used for short, simple operations and are defined using the ‘lambda’.

Syntax:-

lambda arguments : expression

Example:-

```
x = lambda a : a + 10
print(x(5))
```

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

```
x = lambda a, b, c : a + b * c
print(x(5, 6, 2))
```

Using def Function:

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Creating and Using Modules

- Creating and using modules in Python is a great way to organize your code into separate files and promote reusability.
- A module is simply a Python file containing functions, classes, and variables that you can import into other Python scripts.

Create a Python File: Start by creating a new '.py' with a name that is user-defined. (ex: demo.py)

Define Functions/Classes: In the 'demo.py' file, define the functions, classes, or variables that you want to include in the module.

Ex:-

demo.py file

```
def abc(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b
```

Using the Module:

Import the Module: (create a new file name main.py)

```
import demo
result = demo.add(3, 5)
pt = demo.abc(“Ram")
print(result)
print(pt)
```

Access Functions/Variables:

```
module_name.function_name()
module_name.Variable_name()
```

Using Alias:

```
import demo as d
print(d.abc("Ram"))
```

Import Specific Items

```
from demo import abc
print(abc("Ram")
```

Namespace

Namespace refers to a container that holds a set of identifiers (such as variable names, function names, etc.) and provides a way to distinguish between different identifiers that might have the same name but belong to different contexts.

```
from demo import abc
def abc(x, y):
    return x - y
result = abc(3, 5)
print(demo.abc(3,4))
```

__main__ and __name__

- “__name__ and __main__ keywords are used to determine if a Python script is being run as the main program or if it is being imported as a module into another script.

- These keywords play a key role in structuring code that can be both executed directly and used as a module in other programs.

Example:-

```
def say_hello():
        print("Hello from example.py!")
print("This will always be executed")

if  __name__ == "__main__":
    print(“main")
    say_hello()
```

Module Files: In your package directory, create Python module files (‘module.py’, ‘module.py’, etc.) that will contain your functions, class, or variables.

```
# module1.py
def function1():
    print("Function 1 from module1")

# module2.py
def function2():
    print("Function 2 from module2")
```

init.py: The ‘__init__’.py file is executed when the package is imported. You can use it to expose selected contents from your modules for package-level access.

GENERATORS

Generator-Function: A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

```
def m():
    yield 1
    yield 2
    yield 3


for value in m():
    print(value)
```

```
def  num(limit):
    a, b = 0, 1
    count = 0

    while count < limit:
        yield a
        a, b = b, a + b
        count += 1

no = 10
f= fib(no)

for num in f:
    print(num)
```

Generator-Object

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop (as shown in the above program).

```
def f():
    yield 1
    yield 2
    yield 3
x = f()
print(next(x))
print(next(x))
print(next(x))
```

```
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b
x = fib(5)
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

Comprehensions

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined.

- List Comprehensions
- Dictionary Comprehensions
- Set Comprehensions

List Comprehensions:-

List Comprehensions provide an elegant way to create new lists. The following is the basic structure of a list comprehension

variable= [output_exp for var in input_list if (var satisfies this condition)]

```
A = [1, 2, 3, 4, 4, 5, 6, 7, 7]
B  = []
for var in A:
    if var % 2 == 0:
        B.append(var)
print(B)
```

```
A = [1, 2, 3, 4, 4, 5, 6, 7, 7]
B = [i for i in A if i % 2 == 0]
Print(B)
```

Python Dictionary Comprehension

Dictionaries are data types in Python which allows us to store data in **key/value pair**.

```
my_dict = {1: 'apple', 2: 'ball'}
```

```
square_dict = dict()
for num in range(1, 11):
    square_dict[num] = num*num
print(square_dict)
```

```
square_dict = {num: num*num for num in range(1, 11)}
print(square_dict)
```

set comprehension

Set comprehension is a method for creating sets in python using the elements from other iterables like lists, sets, or tuples. Just like we use list comprehension to create lists, we can use set comprehension instead of for loop to create a new set and add elements to it.

newSet= { expression **for** element **in** iterable }

```
myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
newSet = {element*3 for element in myList}
```

Python File Handling

Python offers programmers six access modes to files, and in this article, we'll show you the code needed to create, read, close, and write files in Python. You'll also learn about the various file methods every Python programmer needs to know.

As a computer user, you certainly know about music files, video files, and text files. Well, Python allows you to manipulate these files.

File operations:-

1. Read Only
2. Read and Write
3. Write Only
4. Write and Read
5. Append Only
6. Append and Read

Opening a File: To work with a file, you need to open it first using the open() function. This function returns a file object, which you can use to read or write data.

```
# Opening a file for reading
file = open("example.txt", "r")

# Opening a file for writing
file = open("output.txt", "w")
```

Reading from a File: Once a file is opened for reading, you can use methods like ‘read’ , ‘readline()’, or ‘readlines()’ to read its contents.

```
content = file.read()
line = file.readline()
lines = file.readlines()
```

Writing to a File: When a file is opened for writing, you can use methods like ‘write()’ or ‘writelines()’ to write data into it.

```
file.write("Hello, world!\n")

lines = ["Line 1\n", "Line 2\n"]
file.writelines(lines)
```

Closing a File: It's important to close the file after you're done with it using the ‘close()’ method. This ensures that any changes are saved and resources are released.

```
file.close()
```

'with' Statements: A safer way to work with files is to use the **‘with’** statement, which automatically closes the file when you're done with it, even if an error occurs.

```
with open("example.txt", "r") as file:
    content = file.read()

with open("output.txt", "w") as file:
    file.write("This is some content.")
```

Handling data types in files:-

Text Files: Text files are the simplest form of file storage. When working with text files, you generally convert data to strings before writing and parse strings back to the appropriate data types when reading.

```
# Writing to a text file
with open("data.txt", "w") as file:
    file.write("42\n")
    file.write("3.14\n")
    file.write("Hello, world!\n")

# Reading from a text file
with open("data.txt", "r") as file:
    int_data = int(file.readline().strip())
    float_data = float(file.readline().strip())
    string_data = file.readline().strip()
```

CSV Files: CSV (Comma-Separated Values) files are commonly used to store tabula data. The ‘csv’ module in Python can help you handle data types in CSV files.

```
import csv

with open("data.csv", "w") as file:
    writer = csv.writer(file)
    writer.writerow([42, 3.14, "Hello"])

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    row = next(reader)
    int_data = int(row[0])
    float_data = float(row[1])
    string_data = row[2]
```

JSON Files: JSON (JavaScript Object Notation) files are used to store structured data. The 'json' module in Python is handy for working with JSON files.

```
import json
data = {
    "integer": 42,
    "float": 3.14,
    "string": "Hello"
}
with open("data.json", "w") as file:
    json.dump(data, file)

with open("data.json", "r") as file:
    loaded_data = json.load(file)
    int_data = loaded_data["integer"]
    float_data = loaded_data["float"]
    string_data = loaded_data["string"]
```

Binary Files: For more complex data types and structures, you might need to use binary files. The ‘struct’ module can help you pack and unpack binary data.

```
import struct

with open("data.bin", "wb") as file:
    integer_data = 42
    float_data = 3.14
    file.write(struct.pack("if", integer_data, float_data))

with open("data.bin", "rb") as file:
    packed_data = file.read(8)
    unpacked_data = struct.unpack("if", packed_data)
    int_data = unpacked_data[0]
    float_data = unpacked_data[1]
```

Pickle

Pickle is a built-in Python module used for serializing (converting data to a format that can be stored) and deserializing (converting stored data back to its original format) Python objects. Pickle allows you to save complex data structures, including custom classes and their instances, into a file or a binary stream. It's commonly used for tasks like data persistence, caching, and inter-process communication.

Serialization: Serialization is the process of converting Python objects into a format that can be stored.

```
import pickle

data = {
    "name": "Alice",
    "age": 30,
    "is_student": False
}

with open("data.pickle", "wb") as file:
    pickle.dump(data, file)
```

Deserialization: Deserialization is the process of converting stored data back into Python objects. The ‘pickle. load’ function is used to read serialized data from a file or a binary stream and reconstruct the original object

```
with open("data. pickle", "rb") as file:
    loaded_data = pickle.load(file)

print(loaded_data)
```

```
import pickle,os
def cr():
        f=open("A.dat",'wb')
        ch='Y'
        while ch== "Y":
                ec = int (input("Enter employee code "))
                en = input("enter employee name ")
                sal = float(input("salary"))
                L=[ec,en,sal]
                pickle.dump(L,f)
                ch=input("do you want to continue.... y/n").upper()
        f.close()
def update():
        f=open("A.dat","rb")
        flag=True    g=open("T","wb")
        while flag:
                try:
                        data = pickle.load(f)
                except EOFError:
                        flag = False
                else:
                         data[2]=data[2]+10/100*data[2]
        pickle.dump(data,g)
         f.close()
        g.close()
        os.remove("A.dat")              os.rename("T","A.dat")
cr()
update()
g=open("A.dat","rb")
h=g.read()print(pickle.loads(h)
```

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around the concept of objects, which are instances of classes. OOP promotes the organization of code into reusable and self-contained units, making it easier to manage complexity and create modular, maintainable, and extensible software systems. Here's a brief explanation of some key concepts in OOP:

Class:

1.A class is a blueprint or template for creating objects in object-oriented programming.
2.It defines the structure, attributes, and behaviors that objects of that class will have.
3.Classes are like data types that you define yourself, encapsulating both data and methods.
4.Classes help organize and structure code by grouping related functionality together.
5.You can create multiple objects based on a single class, each with its own unique data but sharing the same methods.

Object:

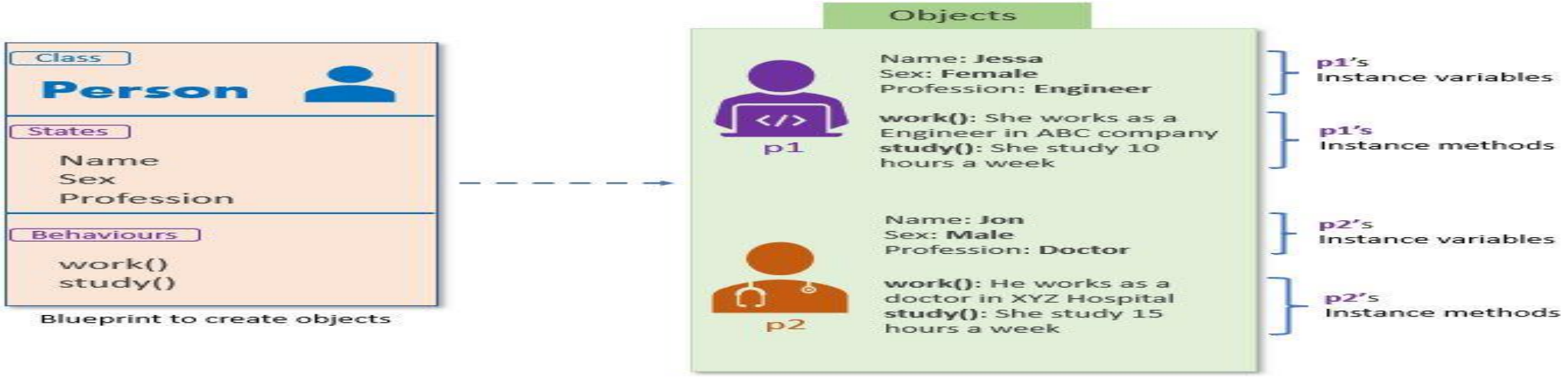
1.An object is an instance of a class, created using the blueprint provided by the class.

2.Objects are concrete entities with specific data values and behaviors defined by the class.

3.Each object has its own set of attributes (data) and can perform actions (methods).

4.Objects are the tangible representation of the abstract concepts described by classes.

5.Objects are used to interact with and manipulate the functionalities defined in the class.



```
# Define a class named 'Car'
class Car:
    def set_details(self, brand, model):
        self.brand = brand
        self.model = model

    def display_details(self):
        print(f"Car details: Brand - {self.brand}, Model-{self.model}")

# Create objects of the 'Car' class
car1 = Car()
car1.set_details("Toyota", "Camry")
car1.display_details()

car2 = Car()
car2.set_details("Honda", "Civic")
car2.display_details()
```

constructor

- A constructor in Python is a special method within a class that automatically initializes the attributes of an object when it's created.
- It has the name ‘___init__’.
- The purpose of the constructor is to set up the object’s initial state by assigning values to its attributes.
- When you create an object from a class, the constructor is automatically called and responsible for ensuring the object starts with the right data.
- This ensures that objects are consistent and properly set up as soon as they're created.

Destructors

A destructor in Python is a special method within a class that gets automatically called when an object is about to be destroyed or deallocated from memory.
It's named ‘__del__’ The purpose of the destructor is to perform any necessary cleanup or resource release operations before the object is removed from memory.

- **Cleanup**: Destructors allow you to release resources, close files, or perform other cleanup tasks associated with an object before it's no longer accessible.
- **Automatic Invocation**: The destructor is automatically invoked when the object is no longer being used or is going out of scope.

```
class FileHandler:
    def __init__(self, f):
        self.f = f
        self.a = open(f, 'r')

    def read_contents(self):
        return self. a.read()

    def __del__(self):
        print(f"Closing the file: {self.filename}")
        self.a.close()

file_reader = FileHandler(“abc.txt")
contents = file_reader.read_contents()
print(contents)
```

Encapsulation

- Encapsulation is one of the four fundamental principles of object-oriented programming (OOP), alongside inheritance, polymorphism, and abstraction.
- It involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit called a class.
- Encapsulation restricts access to the internal details of an object, providing controlled interaction with the object's data and behavior.
- i)Public
- ii) private
- Iii) protected

```
class Car:
    def __init__(self, make, model):
        self.make = make           # Public member
        self._model = model        # Protected member
        self.__year = 2023         # Private member

    def get_year(self):
        return self.__year

    def set_year(self, year):
        if year >= 1900 and year <= 2023:
            self.__year = year

car = Car("Toyota", "Camry")
print(car.make)
print(car._model)
print(car._Car__year)
print(car.get_year())
car.set_year(2022)
print(car.get_year())
```

Polymorphism

Polymorphism is a key concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. In Python, polymorphism is supported by its dynamic typing and duck typing principles.

Inheritance and Method Overriding:

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
make_animal_speak(cat)  # Output: Meow!
```

```
def make_animal_speak(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

make_animal_speak(dog)  # Output: Woof!
```

Duck Typing:

```
class Pen:
    def write(self):
        print("Pen writes.")

class Pencil:
    def write(self):
        print("Pencil writes.")

def use_writing_tool(tool):
    tool.write()

pen = Pen()
pencil = Pencil()

use_writing_tool(pen)     # Output: Pen writes.
use_writing_tool(pencil)  # Output: Pencil writes.
```

Inheritance

Inheritance is another core concept of Object-Oriented Programming (OOP) that allows you to create a new class (subclass or derived class) by inheriting properties and behaviors from an existing class (superclass or base class).

This promotes code reusability and the creation of hierarchies among classes. In Python, you can achieve inheritance using the ‘class’ keyword and specify the base class in parentheses after the derived class name.

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        return f"Make: {self.make}, Model: {self.model}"

class Car(Vehicle):
    def __init__(self, make, model, num_doors):
        super().__init__(make, model
        self.num_doors = num_doors

    def display_info(self):
        base_info = super().display_info()  # Calling the
base class method
        return f"{base_info}, Number of doors:
{self.num_doors}"
```

```
class Bicycle(Vehicle):
    def __init__(self, make, model, frame_size):
        super().__init__(make, model)
        self.frame_size = frame_size

    def display_info(self):
        base_info = super().display_info()
        return f"{base_info}, Frame size: {self.frame_

car = Car("Toyota", "Camry", 4)
bike = Bicycle("Trek", "Mountain Bike", "Medium

print(car.display_info())
print(bike.display_info())
```

MySQL database

Connect to a MySQL database using Python, you can use the "**mysql-connector-python**" library, commonly referred to as MySQL Connector.

This library provides an easy way to interact with MySQL databases, execute SQL queries, and manage database connections.
Here's a basic introduction to using MySQL Connector in Python

Installation

mysql-connector-python library using the following command

```
pip install mysql-connector-python
```

Connecting to the Database

```
import mysql.connector
conn = mysql.connector.connect(
    host='localhost',
    user='username',
    password='password',
    database='dbname'
)
```

Executing Queries

```
# Create a cursor
cursor = conn.cursor()

# Execute SQL queries
cursor.execute('SELECT * FROM users')

# Fetch and print results
results = cursor.fetchall()
for row in results:
    print(row)
```

Close

```
# Close the cursor and connection
cursor.close()
conn.close()
```

Handling Exceptions

```
try:
        #interact with database
except mysql.connector.Error as err:
    print("Error:", err)
finally:
    cursor.close()
    conn.close()
```

```
import mysql.connector

# Replace these with your actual database credentials
host = "localhost"
user = "root"
password = "root"
database = "hjk"

try:
    connection = mysql.connector.connect(
        host=host,
        user=user,
        password=password,
        database=database
    )
    if connection.is_connected():
        print("Connected to the database")
        cursor = connection.cursor()
        for i in range(3):
            a=input("enter ur name")
            b=input("enter ur email id")
            data_to_insert = (a,b)
            insert_query = "INSERT INTO user(name, email) VALUES (%s, %s)"
            cursor.execute(insert_query, data_to_insert)
            connection.commit()
            print("Data inserted successfully")

        # Close the cursor
        cursor.close()

    # Close the connection when done
    connection.close()
    print("Connection closed")

except mysql.connector.Error as e:
    print("Error:", e)
```

```
import mysql.connector

# Replace with your database credentials
db= {
    "host": "your_host",
    "user": "your_user",
    "password": "your_password",
    "database": "your_database"
}
connection = mysql.connector.connect(**db)
cursor = connection.cursor()

dl= []
try:
    query = "SELECT name, id, mobile_number FROM your_table"
    cursor.execute(query)
    results = cursor.fetchall()

    for name, id, mobile_number in results:
        dl.append({
            "name": name,
            "id": id,
            "mobile_number": mobile_number
        })
except mysql.connector.Error as err:
    print("Error:", err)

finally:
    cursor.close()
    connection.close()

# Print the stored data
for entry in data_list:
    print("Name:", entry["name"])
    print("ID:", entry["id"])
    print("Mobile Number:", entry["mobile_number"])
    print("----------------------")
```

Exception Handling

Exceptions:

In Python, an exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. Exceptions can be caused by various factors, such as errors in the code, invalid inputs, or external factors like unavailable resources

Exception Handling:

Exception handling is a mechanism in programming that allows you to gracefully handle exceptions, preventing your program from crashing when an exception occurs. It involves identifying and catching exceptions, then taking appropriate actions to handle them.

1. The try block lets you test a block of code for errors.

2. The except block lets you handle the error.

3. The else block lets you execute code when there is no error.

4. The finally block lets you execute code, regardless of the result of the try- and except blocks.

```
try:
      result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
            print("Error: Division by zero!")
else:
             print("Division successful.")
finally:
            print("End of exception handling.")
```

Exception	Description
IndexError	Raised when the index of a sequence is out of range.
AttributeError	Raised on the attribute assignment or reference fails.
TypeError	Raised when a function or operation is applied to an object of an incorrect type.
IndentationError	Raised when there is an incorrect indentation.
NameError	Raised when a variable is not found in the local or global scope.
SyntaxError	Raised by the parser when a syntax error is encountered.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of a division or module operation is zero.

Name Error:

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

Index Error:-

```
L1=[1,2,3]
L1[3]
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
L1[3]
IndexError: list index out of range
```

1. try…..except…else

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

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

1. try-finally clause

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

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

File and try except program

```
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

1. Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception

```
try:

   You do your operations here

except ExceptionType, Argument:

   You can print value of Argument here...
```

```
def square(var):
        try:
                print(int( var)**2)
        except   ValueError  as  E:
                print("The argument
does not come",E)
square("10")
square("abc")
```

Raising 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.

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

Raising Type Error:-

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

User-defined exception: -

- users can define custom exceptions by creating a new class.
- This exception class has to be derived, either directly or indirectly, from the built-in `Exception` class.
- Most of the built-in exceptions are also derived from this class.

```
class Error(Exception): #new
    """Base class for other exceptions"""
    pass
class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass
class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass
i = 10
```

……2

```
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < i:
            raise ValueTooSmallError
        elif i_num > i:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()
print("Congratulations! You guessed it correctly.")
```

REGULAR EXPRESSION

RegEx Module: Regular expressions (regex) are a powerful tool for pattern matching and manipulation of strings in Python. They allow you to search, match, and manipulate text based on specific patterns.

Python has a built-in package called `re`, which can be used to work with Regular Expressions.Import the `re` module:

```
import re
txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
if x:
  print("YES! We have a match!")
else:
  print("No match")
```

```
pattern = "apple"
text = "I love apples and apple pie."
result = re.findall(pattern, text)
# Output: ['apple', 'apple']
```

```
pattern = "[aeiou]"
text = "Regular expressions are awesome!"
result = re.findall(pattern, text)
# Output: ['e', 'u', 'a', 'e', 'i', 'a', 'e', 'o', 'e']
```

```
import re
a = '^a...s$'
b = 'abyss'
result = re.match(a, b)
if result:
  print("Search successful.")
else:
  print("Search unsuccessful.")
```

```
pattern = "^start"
text = "Starting a new project. This is the start
of something."
result = re.findall(pattern, text)
# Output: ['start']
```

```
pattern = "(apple|banana)"
text = "I like apples and bananas."
result = re.findall(pattern, text)
# Output: ['apple', 'banana']
```

```
text="the colour"
import re
x=re.sub("colour","green colour",text)
print(x)
```

```
import re
a = 'Twelve:12 Eighty nine:89.'
b = '\d+'
result = re.split(b, a)
```

Sockets

Sockets are a fundamental network programming concept that allows communication between different processes or computers over a network, like the Internet. A socket acts as an endpoint for sending or receiving data between two machines. It provides a programming interface to establish a connection, send data, and receive data over the network.

TCP Sockets (Transmission Control Protocol): These provide reliable, connection-oriented communication. When you use TCP sockets, data is sent in a way that ensures it arrives intact and in the correct order. This is commonly used for applications where data integrity and reliability are crucial, such as web browsing, email, and file transfers.

UDP Sockets (User Datagram Protocol): These provide a connectionless, unreliable communication method. Data sent via UDP is not guaranteed to arrive in order or even at all, but it's often used in applications where speed and efficiency are more important than reliability. Examples include online gaming and streaming.

```
import socket

server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
server_socket.bind(("127.0.0.1", 12345))
server_socket.listen()

print("Server is listening...")
client_socket, client_address = server_socket.accept()
print(f"Connection from: {client_address}")

data = client_socket.recv(1024)
print(f"Received: {data.decode()}")

client_socket.close()
server_socket.close()
```

```
import socket

client_socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
client_socket.connect(("127.0.0.1",
12345))

message = "Hello, server!"
client_socket.send(message.encode())

client_socket.close()
```

Thank you..