*Basics of Python Programming *

Introduction

1. Python is a powerful programming language
2. Python was created by Guido van Rossum, and first released on February 20,1991. While you may know the python as a large snake, the name of the Python programming language comes from an old BBC television comedy sketch series called Monty Python's Flying Circus.
3. Supports object-oriented Programming with classes and multiple inheritance.
4. Comes with large standard library.
5. Includes an IDLE.IDLE is Python's Integrated Development and Learning Environment.IDLE is a very small and simple cross-platform IDE
   that is included free with Python and is released under the open-source Python Software Foundation License. Anaconda and Miniconda include IDLE. Runs on Mac OS X, Windows, Linux as Freeware.
6. Specially used in data science,machine learning to solve daily life
   real-time problems and providing the solutions.
7. Python is a high-level,intrepeted,interactive and object oriented scripting langauge.

History of Python language:

1.Python was developed by "Guido Van Rossum" in late 1980's in National Research Institute for Mathematics and Computer Science in Netherlands.

2.Python was derived from many other languages including ABC,Modulo-3,C,C++,Algol=68,SmallTalk,Unix shell and other scripting languages.

Why python?

1. **Easy to Use:** Python is very easy to use and high level language. Thus it is programmer-friendly language.
2. **Expressive Language:** Python language is more expressive. The sense of expressive is the code is easily understandable.

3. **Interpreted Language:** Python is an interpreted language i.e., interpreter executes the code line by line at a time. This makes debugging easy and thus suitale for beginners.
4. **Cross-platform language:** Python can run equally on different platforms such as Windows, Linux, Unix, Macintosh etc. Thus, Python is a portable language.
5. **Free and open source:** Python language is freely available.
6. **Object-Oriented Language:** Python supports object oriented language. Concepts of classes and objects comes into existence.
7. **Large Standard Library:** Python has a large and broad library.
8. **GUI Programming:** GUI can be developed using Python.
9. **Integrated:** It can be easily integrated with languages like C, C++, Java etc.

Applications of python

1. Web Development
2. Game Development
3. Machine Learning and Artificial Intelligence
4. Data Science and Data Visualization
5. Desktop GUI
6. Web Scraping Applications
7. Business Applications
8. Audio and Video Applications
9. CAD Applications
10. Embedded Applications
11. Web Scraping
12. Computer Vision and Image Processing etc.,

Keywords in Python

Here are the key points related to keywords in Python:

1. **Definition**: Keywords in Python are reserved words that cannot be used as ordinary identifiers. They are used to define the syntax and structure of the Python language.

2. **Immutability**: Keywords are immutable. This means their meaning and definition can't be altered.
3. **Case Sensitivity**: Keywords are case-sensitive. For example, `True` is a valid keyword, but `true` is not.
4. **Total Number**: As of Python 3.9, there are 35 keywords.
5. **List of Keywords**: The complete list of Python keywords are `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`.
6. **Special Keywords**: `async` and `await` are used for handling asynchronous processing, and they became keywords in Python 3.7.
7. **Usage**: Each keyword has a specific meaning and usage in Python programming. For instance, `def` is used for defining functions, `if` is used for making conditional statements, `for` and `while` are used for loops, `class` is used for defining a class, and so on.
8. **Identifying Keywords**: You can get the list of all keywords in Python by using the following code:

```
# instal package "keyword" to check how many keywords are there
# pip install keyword

# load package
import keyword

# display all keywords
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
```

```
# To check current working directory
%pwd
```

```
'C:\\Users\\Dell\\Desktop\\iNeuron\\Sessions\\2021May2023'
```

```
none = 10
```

```
# Incorrect use of keywords as variable names

class = 10     # 'class' is a reserved keyword
return = 5     # 'return' is a reserved keyword
if = "hello"   # 'if' is a reserved keyword
```

```
else = 3.14    # 'else' is a reserved keyword

print(class, return, if, else)



# Correct use of variable names

class_number = 10
return_value = 5
if_string = "hello"
else_number = 3.14

print(class_number, return_value, if_string, else_number)
```

Identifiers

Here are the key points related to identifiers in Python:

1. **Definition**: An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.
2. **Syntax**: Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
3. **No digits:** They must start with a letter or the underscore character, but not with a digit.
4. **Case-Sensitive**: Identifiers in Python are case-sensitive. For example, `myVariable` and `myvariable` are two different identifiers in Python.
5. **No Special Characters**: Identifiers cannot have special characters such as `!`, `@`, `#`, `$`, `%`, etc.
6. **Reserved Words**: Python keywords cannot be used as identifiers. Words like `for`, `while`, `break`, `continue`, `in`, `elif`, `else`, `import`, `from`, `pass`, `return`, etc. are reserved words. You can view all keywords in your current version by typing `help("keywords")` in the Python interpreter.
7. **Unlimited Length**: Python does not put any restriction on the length of the identifier. However, it's recommended to keep it within a reasonable size, to maintain readability and simplicity in the code.
8. **Private Identifiers**: In Python, if the identifier starts with a single underscore, it indicates that it is a non-public part of the class, module, or function. This is just a convention and Python doesn't enforce it. If it starts with two underscores, it's a strongly private identifier. If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

9. **Non-ASCII Identifiers**: Python 3 allows the use of non-ASCII letters in the identifiers. This means you can use letters like é, ñ, ö, я, etc. in your identifiers if you wish.

```
num1 = 5
_value = 10
text_string = 'Hello, Everyone!'
print(num1, _value, text_string)
```

```
5 10 Hello, Everyone!
```

```
_1a = 10
_1b = 20
c = _1a + _1b
print(c)
```

```
30
```

```
my_var = 'lowercase'
MY_VAR = 'UPPERCASE'
print(my_var, MY_VAR)
```

```
lowercase UPPERCASE
```

Comments in python

Here are some key points about comments in Python:

1. **Definition:** A comment in Python is a piece of text in your code that is not executed. It's typically used to explain what the code is doing or leave notes for developers who will be reading or maintaining the code.
2. **Single-Line Comments:** Python uses the hash symbol (#) to denote a comment. Any text following the # on the same line will be ignored by the Python interpreter.
3. **Multi-Line Comments:** Python does not have a specific syntax for multi-line comments. Developers typically use a single # for each line to create multi-line comments. Alternatively, multi-line strings using triple quotes (''' or """) can also be used as multi-line comments because any string not assigned to a variable is ignored by Python.
4. **Docstrings or documentation strings:** Docstrings are a type of comment used to explain the purpose of a function or a class. They are created using triple quotes and are placed immediately after the definition of a function or a class. Docstrings can span multiple lines and are accessible at runtime using the .__doc__ attribute.

```
# This is a comment in Python
x = 5  # This is an inline comment


# This is a multi-line comment
# We are adding two numbers a and b.
a = 5
b = 3
c = a + b
print(c)
```

```
 8
```

```
# Docstring Example
def add_numbers(a, b):
    """
    This function adds two numbers and returns the result.

    Parameters:
    a (int): The first number
    b (int): The second number

    Returns:
    int: The sum of the two numbers
    """
    return a + b



# Docstring Example
def add_numbers1(a, b):
    '''
    This function adds two numbers and returns the result.

    Parameters:
    a (int): The first number
    b (int): The second number

    Returns:
    int: The sum of the two numbers
    '''
    return a + b



# You can access this docstring using the .__doc__ attribute.
# Here's how:
print(add_numbers1.__doc__)
```

```

    This function adds two numbers and returns the result.

    Parameters:
    a (int): The first number
    b (int): The second number
```

```
        Returns:
        int: The sum of the two numbers


#single line comment--->shows the single line comment.
print('hello my dear friends')


"""
multi line comment
How are you all
Have a nice day guys
"""
print('Bye Everyone')


# Incorrect use of identifiers

1variable = 10      # Identifier starts with a digit
$second = 20        # Identifier contains special character
third variable = 30 # Identifier contains a space
for = 40            # Identifier is a reserved keyword

print(1variable, $second, third variable, for)
```

```
  File "<ipython-input-74-b9f72a3509ec>", line 3
    1variable = 10      # Identifier starts with a digit
     ^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

```
# Correct use of identifiers

variable1 = 10
second_variable = 20
third_variable = 30
variable_for = 40

print(variable1, second_variable, third_variable, variable_for)
```

```
10 20 30 40
```

Indentation

Indentation

1. **Importance**: In Python, indentation is not just for readability. It's a part of the syntax and is used to indicate a block of code.
2. **Space Usage**: Python uses indentation to define the scope of loops, functions, classes, etc. The standard practice is to use four spaces for each level of indentation, but you can use any number of spaces, as long as the indentation is consistent within a block of code.
3. **Colon**: Usually, a colon (`:`) at the end of the line is followed by an indented block of code. This is common with structures like `if`, `for`, `while`, `def`, `class`, etc.

```
def say_hello():
    print("Hello, Everyone!")


# Incorrect use of indentation

if True:
print("This is True!") # Error: expected an indented block

for i in range(3):
print(i) # Error: expected an indented block

def hello():
print("Hello, World!") # Error: expected an indented block

while False:
print("This won't print") # Error: expected an indented block



# Correct use of indentation

if True:
    print("This is True!")

for i in range(3):
    print(i)

def hello():
    print("Hello, World!")

while False:
    print("This won't print")
```

Statements

1. **Definition:** A statement in Python is a logical instruction that the Python interpreter can read and execute. In general, a statement performs some action or action.

2. **Types:** Python includes several types of statements including assignment statements, conditional statements, looping statements, etc.

```
# Conditional Statement
x = 2
if x > 0:
    print("Positive number")
```

```
#Looping statement
for i in range(5):
    print(i)
```

Multi-Line Statements: In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (`\`), or within parentheses `()`, brackets `[]`, braces `{}`, or strings. You can also write multiple statements on a single line using semicolons (;)

```
# Multi-Line Statements
# Using line continuation character
s = 1 + 2 + 3 + \
    4 + 5

# Using parentheses
s = (1 + 2 + 3 +
     4 + 5)

# Multiple Statements on a Single Line
x = 5; y = 10; print(x + y)
```

```
15
```

Variables

1. **Definition**: In Python, a variable is a named location used to store data in memory.
2. **Declaration and Assignment**: Variables are declared by writing the variable name and assigning it a value using the equals sign (`=`). For example:

```
a = 5
st = "hello"
```

Dynamic Typing: Python is dynamically typed, which means that you don't have to declare the type of a variable when you create one. You can even change the type of data held by a variable at any time.

```
x = 5   # x is an integer
x = "Hello"  # now x is a string
```

Checking the type of a value

To know the exact type of any value, Python provides an in-built method called type.

```
type('Hello World')
```

```
type(123)
```

```
type(23.4)
```

```
type(True)
```

Python Data types:

1.Integer data type

2.Float point numbers

3.Strings data type

4.Boolean data type

note:Python accepts all types datas as listed above.

Data types in Python

1. **Integers**: Integers are whole numbers, without a fractional component. They can be positive or negative.

```
x = 10
y = -3
```

The int function

```
The int function converts a string or a number into an integer.
The int function removes everything after the decimal point.
```

```
int(12.346)
```

```
int('123')
```

2. **Floats**: Floats represent real numbers and are written with a decimal point.

```
x = 10.0
y = -3.14
```

The float function

The float function converts a string into a floating-point number.

```
float('10.23')
```

3. **Strings**: Strings in Python are sequences of character data. They are created by enclosing characters in quotes.

```
s = "Hello, World!"
print(s)
```

```
Hello, World!
```

The str function

The str function is used to convert a number into a string.

```
str(12)
```

4. **Boolean Type**

The boolean type is represented in Python as bool. It is a primitive data type haivng True or False

```
type(True)
```

Python Data Structures

In python we have four types of data structures

These data structures are very useful and important while working with advance topics such as machine learning ,data science.

1.List

2.Tuple

3.Dictionary

4.Sets

Lists

Definition: A list in Python is an ordered collection (also known as a sequence) of items. Lists are similar to arrays in other languages, but with additional functionality.

Mutable: Lists are mutable, which means their elements can be changed (added, modified, or deleted) after they are created.

Creation: A list is created by placing items (elements) inside square brackets [], separated by commas.

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

```
['apple', 'banana', 'cherry']
```

Heterogeneous Elements: A list can contain elements of different types: integers, floats, strings, and even other lists or tuples.

```
mixed_list = [1, "Alice", 3.14, [5, 6, 7]]
print(mixed_list)
```

```
[1, 'Alice', 3.14, [5, 6, 7]]
```

Indexing and Slicing: Lists support indexing and slicing to access and modify their elements. Python list indices start at 0.

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0])  # 1
print(my_list[1:3])  # [2, 3]
my_list[0] = 10  # change the first element to 10
print(my_list)
```

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

Tuples

1. **Definition**: A tuple in Python is similar to a list. It's an ordered collection of items.
2. **Immutable**: The major difference from lists is that tuples are immutable, which means their elements cannot be changed (no addition, modification, or deletion) after they are created.
3. **Creation**: A tuple is created by placing items (elements) inside parentheses `()`, separated by commas.

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

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

Heterogeneous Elements: Like lists, a tuple can also contain elements of different types: integers, floats, strings, and even other tuples or lists.

```
my_tuple = (1, 2, 3, "Tom", [5, 6, 7])
print(my_tuple)
```

```
(1, 2, 3, 'Tom', [5, 6, 7])
```

Indexing and Slicing: Tuples also support indexing and slicing, like lists, but you cannot modify their elements.

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[0])  # 1
print(my_tuple[1:3])  # (2, 3)
```

```
1
(2, 3)
```

Dictionaries

1. **Definition**: A dictionary in Python is an unordered collection of items. Each item stored in a dictionary has a key and value, making it a key-value pair.
2. **Mutable**: Dictionaries are mutable, which means you can change their elements. You can add, modify, or delete key-value pairs from a dictionary.
3. **Creation**: A dictionary is created by placing items (key-value pairs) inside curly braces `{}`, separated by commas. Each item is a pair made up of a key and a value, separated by a colon `:`.

```
person = {"name": "Alice", "age": 25}
print(person)
```

```
{'name': 'Alice', 'age': 25}
```

Heterogeneous Elements: Keys and values in a dictionary can be of any type. Values can be heterogeneous, but keys should be of an immutable type (like string, number, or tuple).

```
my_dict = {1: "Tom", "age": 23, (1, 3): [4, 5, 3]}
print(my_dict)
```

```
{1: 'Tom', 'age': 23, (1, 3): [4, 5, 3]}
```

Unique Keys: Each key in a dictionary should be unique. If a dictionary is created with duplicate keys, the last assignment will overwrite the previous ones.

```
my_dict = {"name": "Alice", "name": "Bob"}
print(my_dict)  # {'name': 'Bob'}
```

```
{'name': 'Bob'}
```

Accessing Values: You can access a value in a dictionary by providing the corresponding key inside square brackets `[]`.

```
my_dict = {"name": "Tom", "age": 21}
print(my_dict["name"])  # Tom
```

```
Alice
```

Updating Values: You can update the value for a particular key by using the assignment operator `=`.

```
my_dict = {"name": "Tom", "age": 21}

my_dict["age"] = 20

print(my_dict)  # {'name': 'tom', 'age': 20}
```

```
{'name': 'Tom', 'age': 20}
```

Adding and Deleting Key-Value Pairs: You can add a new key-value pair simply by assigning a value to a new key. You can delete a key-value pair using the `del` keyword.

```
my_dict = {"name": "Tom", "age": 20}

my_dict["city"] = "New York"  # adding a new key-value pair

del my_dict["age"]  # deleting a key-value pair

print(my_dict)  # {'name': 'Tom', 'city': 'New York'}
```

```
{'name': 'Alice', 'city': 'New York'}
```

Sets: A set is an unordered collection of items where every element is unique.

```
colors = {"red", "green", "blue"}
print(colors)
```

```
{'red', 'blue', 'green'}
```

To determine the type of a variable, you can use the `type()` function:

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

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

```
# Incorrect use of data types

# Trying to add a string and an integer
result1 = "5" + 3  # Error: must be str, not int

# Trying to access a non-existing index of a list
my_list = [1, 2, 3]
result2 = my_list[5]  # Error: list index out of range

# Trying to change a value in a tuple
my_tuple = (1, 2, 3)
```

```
my_tuple[1] = 4  # Error: 'tuple' object does not support item assignment

# Trying to access a non-existing key in a dictionary
my_dict = {"one": 1, "two": 2}
result4 = my_dict["three"]  # Error: 'three' key not found

print(result1, result2, my_tuple, result4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-18b47f82609c> in <cell line: 4>()
      2
      3 # Trying to add a string and an integer
----> 4 result1 = "5" + 3  # Error: must be str, not int
      5
      6 # Trying to access a non-existing index of a list

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

SEARCH STACK OVERFLOW

```
# Correct use of data types

# Convert the integer to a string before adding
result1 = "5" + str(3)

# Check if the index exists before accessing
result2 = my_list[2] if len(my_list) > 2 else None

# Tuples are immutable, create a new one if you need to change a value
my_tuple = (1, 4, 3)

# Check if the key exists before accessing
result4 = my_dict.get("three", None)

print(result1, result2, my_tuple, result4)
```

```
53 3 (1, 4, 3) None
```

Standard Input and Output

Standard Output

- This is typically the terminal (or the console) where the program is run. When a program wants to output some information, it will typically print to the standard output.
- Python provides `print()` function to send data to standard output. Here is an example:

The print function is used to display the contents on the screen. The syntax of print function is:

```
    print(argument)
```

The argument of the print function can be a value of any type int, str, float etc. It can be a value stored in a variable.

```
print("Welcome to Python class")
# In this example, the string "Welcome to iNeuron" is sent to the standard output.
# Usually your terminal or console where you are running the program.
```

```
     Welcome to Python class
```

```
print('Hello Welcome to Python Programming')
```

```
print(10000)
```

```
print('Display String Demo')
```

```
A=10
B=20
print('The value of A is',A,'and the value of B is',B)
print(A,'is the value of A')
```

Escape Character

Using Escapre character we can print the special characters as it is in print function.

Escape sequences	*Meaning*
\'	single quote
\"	double quote
\\	backslash
\n	new line
\t	tab
\b	backspace

```
print("The flight attendant asked, " May I see your boarding pass?" ")
```

```
print("The flight attendant asked, \" May I see your boarding pass?\" ")
```

f-String

When you're formatting strings in Python, you're probably used to using the format() method.

But in Python 3.6 and later, you can use f-Strings instead. f-Strings, also called formatted string literals, have a more succinct syntax and can be super helpful in string formatting.

Strings in Python are usually enclosed within double quotes ("") or single quotes ('). To create f-strings, you only need to add an f or an F before the opening quotes of your string.

When using f-Strings to display variables, you only need to specify the names of the variables inside a set of curly braces {}. And at runtime, all variable names will be replaced with their respective values.

- The `f` before the string in `print(f"Hello, {name}!")` is used to denote a formatted string literal, often called f-string for short.
- F-strings were introduced in Python 3.6 as a new way to format strings. They are prefixed with 'f' and are followed by a string literal enclosed in quotes. The expression within the curly braces `{}` gets evaluated and inserted into the string.

```
language = "Python"
school = "School"
print(f"I'm learning {language} from {school}.")
```

```
I'm learning Python from School.
```

```
print("I'm learning",language, "from",school)
```

```
I'm learning Python from School
```

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

```
num1 = 83
num2 = 9
print(f"The product of {num1} and {num2} is {num1 * num2}.")
```

```
The product of 83 and 9 is 747.
```

Standard Input:

- This is usually the keyboard, but it can also be data coming from a file or another program.
- Python provides the `input()` function to read data from standard input. Here is an example:

```
# Take input from the user for two numbers
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))

# Calculate the sum of the two numbers
sum_of_numbers = num1 + num2

# Print the result
```

```
print(f"The sum of {num1} and {num2} is {sum_of_numbers}.")
```

```
Enter the first number: 5
Enter the second number: 7
The sum of 5 and 7 is 12.
```

```
Maths = 90
English = 85
print("Your marks in Maths:", Maths, "and in English:", English)
```

```
Your marks in Maths: 90 and in English: 85
```

```
Maths = 90
English = 85

print("Your marks in Maths:", Maths, "\nYour marks in English:", English)
```

```
Your marks in Maths: 90
Your marks in English: 85
```

```
#The input() function produces only string. That means, The return type of input() is alwa

#So we may use the int() function or float() function etc to convert the read string from
```

```
print(' Please Enter Number')
Num1 = input()
print('Num1 = ',Num1)
print(type(Num1))
print(' Converting type of Num1 to int ')
Num1 = int(Num1)
print(Num1)
print(type(Num1))
```

Write a Python program that prompts the user to enter two numbers. The program should then perform addition, subtraction, multiplication, and division on these numbers and display the results.

```
# Prompt the user to enter two numbers
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Perform arithmetic operations
sum = num1 + num2
difference = num1 - num2
product = num1 * num2

# Handle division carefully as division by zero is undefined
if num2 != 0:
    quotient = num1 / num2
```

```
else:
    quotient = "Undefined (division by zero)"

# Print the results
print(f"The sum of {num1} and {num2} is: {sum}")
print(f"The difference between {num1} and {num2} is: {difference}")
print(f"The product of {num1} and {num2} is: {product}")
print(f"The quotient of {num1} and {num2} is: {quotient}")
```

The eval() function

It takes a string as a parameter and returs it as if it is a python expression. For example, the statement eval('print("Hello")') in Python will actually run the statement print("Hello").

Write a program to display details entered by a user, i.e., name, age, gender and height.

```
Name = (input('Enter Name: '))
Age = eval(input('Enter Age: '))
Gender = (input('Enter Gender: '))
Height = eval(input('Enter Height: '))

print('Name: ', Name)
print('Age: ', Age)
print('Gender: ', Gender)
print('Height: ', Height)
```

Operators

Various Operators in Python are:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic Operators: Used to perform mathematical operations.

```
a = 10
b = 3

print(a + b)  # Addition, Output: 13
```

```
print(a - b)  # Subtraction, Output: 7
print(a * b)  # Multiplication, Output: 30
print(a / b)  # Division, Output: 3.3333333333333335
print(a // b) # Floor Division, Output: 3
print(a % b)  # Modulus, Output: 1
print(a ** b) # Exponent, Output: 1000
```

```
13
7
30
3.3333333333333335
3
1
1000
```

Assignment Operators: Used to assign values to variables.

a = 10 # Assigns value 10 to a print(a)

a += 5 # Same as a = a + 5, Output: 15 print(a)

a -= 3 # Same as a = a - 3, Output: 12 print(a)

a *= 2 # Same as a = a * 2, Output: 24 print(a)

a /= 6 # Same as a = a / 6, Output: 4.0 print(a)

Comparison Operators: Used to compare two values.

```
a = 10
b = 20

print(a == b)  # Equal to, Output: False
print(a != b)  # Not equal to, Output: True
print(a > b)   # Greater than, Output: False
print(a < b)   # Less than, Output: True
print(a >= b)  # Greater than or equal to, Output: False
print(a <= b)  # Less than or equal to, Output: True
```

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

Logical Operators: Used to combine conditional statements.

```
a = True
b = False
```

```
print(a and b) # Logical AND, Output: False
print(a or b)  # Logical OR, Output: True
print(not a)   # Logical NOT, Output: False
```

```
False
True
False
```

Bitwise Operators: Used to perform bitwise calculations on integers.

AND Operator `&`: Compares each bit of the first operand with the corresponding bit of the second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.

```
a = 10       # in binary: 1010
b = 4        # in binary: 0100
result = a & b  # result is 0 (in binary: 0000)
print(result)
```

```
0
```

OR Operator `|`: Compares each bit of the first operand with the corresponding bit of the second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.

```
a = 10       # in binary: 1010
b = 4        # in binary: 0100
result = a | b  # result is 14 (in binary: 1110)
print(result)
```

```
14
```

NOT Operator `~`: Inverts all the bits of the operand. Every 0 is changed to 1, and every 1 is changed to 0.

```
a = 10         # in binary: 1010
result = ~a    # result is -11 (in binary: -1011)
print(result)
```

```
-11
```

XOR Operator `^`: Compares each bit of the first operand with the corresponding bit of the second operand. If one of the bits is 1 (but not both), the corresponding result bit is set to 1.

Otherwise, the result bit is set to 0.

```
a = 10       # in binary: 1010
b = 4        # in binary: 0100
result = a ^ b  # result is 14 (in binary: 1110)
print(result)
```

```
14
```

Right Shift Operator `>>` : Shifts the bits of the number to the right by the number of bits specified. In other words, right-shifting an integer “a” with an integer “b” denoted as ‘(a>>b)‘ is equivalent to dividing a with 2^b

```
a = 10        # in binary: 1010
result = a >> 2  # result is 2 (in binary: 0010)
print(result)
```

```
2
```

Left Shift Operator `<<` : Shifts the bits of the number to the left by the number of bits specified. In other words, left-shifting an integer 'a' with an integer 'b' denoted as (a<< b) is equivalent to multiplying 'a' with 2^b (2 raised to power b)

.

```
a = 10        # in binary: 1010
result = a << 2  # result is 40 (in binary: 101000)
print(result)
```

```
40
```

Remember, bitwise operations are only applicable to integers.

Membership Operators: Used to test whether a value or variable is found in a sequence (string, list, tuple, set, and dictionary).

```
list = [1, 2, 3, 4, 5]
print(1 in list)    # Output: True
print(6 not in list) # Output: True
```

```
True
True
```

Identity Operators: Used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

```
a = 5
b = 5
print(a is b)
```

```
True
```

Let us solve few problems:

```
# Using Arithmetic Operators
num1 = 10
num2 = 5

add = num1 + num2
print("The addition of num1 and num2 is:", add)

sub = num1 - num2
print("The subtraction of num1 and num2 is:", sub)

mul = num1 * num2
print("The multiplication of num1 and num2 is:", mul)
```

```
The addition of num1 and num2 is: 15
The subtraction of num1 and num2 is: 5
The multiplication of num1 and num2 is: 50
```

```
# Write this code using Assignment operators
# Using Assignment Operators
num1 = 10
num2 = 5

num1 += num2  # equivalent to num1 = num1 + num2
print("The addition of num1 and num2 is:", num1)

num1 = 10  # reset num1
num1 -= num2  # equivalent to num1 = num1 - num2
print("The subtraction of num1 and num2 is:", num1)

num1 = 10  # reset num1
num1 *= num2  # equivalent to num1 = num1 * num2
print("The multiplication of num1 and num2 is:", num1)
```

```
The addition of num1 and num2 is: 15
The subtraction of num1 and num2 is: 5
The multiplication of num1 and num2 is: 50
```

Write a program to calculate the area of a rectangle

```
length = 10
breadth = 20
area = length * breadth
print('Area of Rectangle is = ',area)
print('The area of the above rectangle is',area)
print('The area of rectangle having length',length,'and breadth',breadth,'is',area)



length=int(input('Enter the length'))
breadth= int(input('Enter the breadth'))
area = length * breadth
print('Area of Rectangle is = ',area)
print('The area of the above rectangle is',area)
print('The area of rectangle having length',length,'and breadth',breadth,'is',area)
```