UNIT-1

What is Programming ?

- Programming is the process of instructing a computer to perform specific tasks by providing a set of instructions written in a programming language. It involves creating algorithms and implementing logic to solve problems or automate processes.

- Python was created in the late 1980s by Guido van Rossum.
- The first official Python release, Python 0.9.0, was released in February 1991.

Design Philosophy

- Python's design philosophy emphasizes readability, making it easy for programmers to express ideas in code.
- It follows the "Zen of Python," a set of guiding principles that promote simplicity, clarity, and beauty in code

Batteries included

- Python is often referred to as a "batteries-included" language, meaning it comes with a comprehensive standard library.
- The standard library provides modules and packages for a wide range of tasks, reducing the need for external libraries.

General Purpose

- Python is a general-purpose language, meaning it can be used for various applications, from web development and automation to scientific research and machine learning.
- Its versatility makes it a top choice for solving a wide range of problems.

Libraries/Community

- Python has a vast and active community of developers who contribute to open-source libraries and frameworks.
- Popular libraries like NumPy, Pandas, TensorFlow, Django, and Flask extend Python's capabilities for specific domains.
- The Python community is known for its support and resources, making it easier to find solutions and assistance when working with Python.

Why Python for Data Science?

Easy to learn

- Python is renowned for its simplicity and readability, making it an accessible language for both beginners and experienced programmers.
- The clean and intuitive syntax reduces the learning curve, allowing data scientists to focus on data analysis rather than wrestling with the language.

Proximity with Maths

- Python offers a wide array of libraries and tools that are specifically designed for data analysis and scientific computing.
- Libraries like NumPy, SciPy, and pandas provide efficient and easy-to-use data structures and functions for numerical and statistical operations.
- Python's compatibility with mathematical operations and libraries makes it a natural fit for data science tasks.

Community

- Python has a thriving and active community of data scientists, analysts, and developers.
- This community contributes to an extensive ecosystem of libraries and resources, including data visualization tools (Matplotlib, Seaborn), machine learning frameworks (Scikit-Learn, TensorFlow), and data manipulation libraries (Pandas).
- The availability of resources, forums, and tutorials makes it easy for data scientists to find help, collaborate, and stay updated with the latest developments in the field.

1. Python Output

In Python, when you write something like vishal(), you are indeed calling a function named vishal. The parentheses () are used to indicate that the function is being invoked or called. To be more specific, vishal() is the syntax used to call a function without passing any arguments.

In [14]:
```
1 help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdou
t.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In [7]:
```
1 # Python is a case sensitive language
2 print('Hello World')
```

```
Hello World
```

In [8]:
```
1 print('INDIA')
```

```
INDIA
```

In [9]:
```
1 print(INDIA)
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
~\AppData\Local\Temp\ipykernel_7452\1453359074.py in <module>
----> 1 print(INDIA)

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

In [10]:
```
1 print(7)
```

```
7
```

In [11]:
```
1 print(7.7)
```

```
7.7
```

In [12]:
```
1 print(True)
```

```
True
```

In [13]:
```
1 print('hello',1,4,5.5,True,False)
```

```
hello 1 4 5.5 True False
```

In [15]:
```
print('hello',1,4,5.5,True,False,sep="v")
```

```
hellov1v4v5.5vTruevFalse
```

In [16]:
```
print('hello',1,4,5.5,True,False,sep="/")
```

```
hello/1/4/5.5/True/False
```

In [17]:
```
print("hello")
print("world")
```

```
hello
world
```

In [18]:
```
print("hello",end=".")
print("world")
```

```
hello.world
```

In [20]:
```
print('hello''world')
```

```
helloworld
```

2.Datatype

In [22]:
```
# Integer
print(8)
# 1*10^308
print(1e308)
```

```
8
1e+308
```

In [23]:
```
print(1e308)
```

```
1e+308
```

In [26]:
```
print(-5e307)
```

```
-5e+307
```

In [29]:
```
# Decimal/Float
print(8.55)
print(1.7e308)
```

```
8.55
1.7e+308
```

In [30]:
```
# Decimal/Float
print(8.55)
print(1.7e309)
```

```
8.55
inf
```

In [31]:
```
# Boolean
print(True)
print(False)
```

```
True
False
```

In [32]:
```
print(5==5)
```

```
True
```

In [33]:
```
# Text/String
print('Hello World')
```

```
Hello World
```

In [34]:
```
# complex
print(5+6j)
```

```
(5+6j)
```

In [35]:
```
# List-> C-> Array
print([1,2,3,4,5])
```

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

In [36]:
```
# Tuple
print((1,2,3,4,5))
```

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

In [37]:
```
# Sets
print({1,2,3,4,5})
```

```
{1, 2, 3, 4, 5}
```

In [39]:
```
# Dictionary
print({'name':'Vishal','gender':'Male','weight':77})
```

```
{'name': 'Vishal', 'gender': 'Male', 'weight': 77}
```

In [54]:
```
# type
print(type([1,2,3]))
print(type(7))
print(type((7)))
print(type((7,)))
print(type(7.7))
print(type("vishal"))
print(type({5,1}))
print(type({'name':'Vishal','gender':'Male','weight':77}))
print(type(5+6j))
```

```
<class 'list'>
<class 'int'>
<class 'int'>
<class 'tuple'>
<class 'float'>
<class 'str'>
<class 'set'>
<class 'dict'>
<class 'complex'>
```


3.variable

A variable is a named storage location in a program's memory that can hold and manipulate data.

- in python,variable do not need to be decleared with any perticular data type

In [56]:
```
# Static Vs Dynamic Typing
# Static Vs Dynamic Binding
# stylish declaration techniques
```

In [59]:
```
name = 'Vishal'
print(name)
a = 5
b = 6
print(a + b)
```

```
Vishal
11
```

Dynamic Typing

a = 5

Static Typing

int a = 5

In [61]:
```
# Dynamic Binding
a = 5
print(a)
a = 'nitish'
print(a)
```

```
5
nitish
```


Static Binding

int a = 5

In [63]:
```
a = 1
b = 2
c = 3
print(a,b,c)
```

```
1 2 3
```

In [64]:
```
a,b,c = 1,2,3
print(a,b,c)
```

```
1 2 3
```

In [65]:
```
a=b=c= 5
print(a,b,c)
```

```
5 5 5
```

Comments

In [2]:
```
# this is a comment
# second line
a = 7
b = 6 # like this
# second comment
print(a+b)
"""multi
line
comment"""
```

```
13
```

Out[2]:
```
'multi \nline\ncomment'
```

4. Keywords & Identifiers

Reserve keywords in python

```
Python Keywords are some predefined and reserved words in Python
that have special meanings. Keywords are used to define the syntax
of the coding. The keyword cannot be used as an identifier,
function, or variable name. All the keywords in Python are written
in lowercase except True,None and False.
```

In [1]:
```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more hel
p.

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



```
Identifier is a user-defined name given to a variable, function,
class, module, etc. The identifier is a combination of character
digits and an underscore. They are case-sensitive i.e., 'num' and
'Num' and 'NUM' are three different identifiers in python.
```

Rules for Naming Python Identifiers

- It cannot be a reserved python keyword.
- It should not contain white space.
- It can be a combination of A-Z, a-z, 0-9, or underscore.
- It should start with an alphabet character or an underscore (_).
- It should not contain any special character other than an underscore (_).

```
Valid identifiers:

var1
_var1
_1_var
var_1
Invalid Identifiers

!var1
1var
1_var
var#1
var 1
```

In [77]:
```
!var=1
```

```
'var' is not recognized as an internal or external command,
operable program or batch file.
```

In [78]:
```
1var=7
```

```
  File "C:\Users\VISHAL\AppData\Local\Temp\ipykernel_7452\2199671680.py", line 1
    1var=7
     ^
SyntaxError: invalid syntax
```


Camel Case

- Definition: Each word starts with a capital letter except for the first word.
- Example: myVariableName

Snake Case

- Definition: Words are separated by underscores.
- Example: my_variable_name

Pascal Case

- Definition: Similar to camel case but starts with a capital letter.
- Example: MyVariableName

5. User Input

In [79]:
```
help(input)
```

```
Help on method raw_input in module ipykernel.kernelbase:

raw_input(prompt='') method of ipykernel.ipkernel.IPythonKernel instance
    Forward raw_input to frontends

    Raises
    ------
    StdinNotImplementedError if active frontend doesn't support stdin.
```

In [80]:
```
help(eval)
```

```
Help on built-in function eval in module builtins:

eval(source, globals=None, locals=None, /)
    Evaluate the given source in the context of globals and locals.

    The source may be a string representing a Python expression
    or a code object as returned by compile().
    The globals must be a dictionary and locals can be any mapping,
    defaulting to the current globals and locals.
    If only globals is given, locals defaults to it.
```

In [81]:
```
x=input("enter number")
print(x)
print(type(x))
```

```
enter number5
5
<class 'str'>
```

In [82]:
```
# take input from users and store them in a variable
fnum = int(input('enter first number'))
snum = int(input('enter second number'))
#print(type(fnum),type(snum))
# add the 2 variables
result = fnum + snum
# print the result
print(result)
print(type(fnum))
```

```
enter first number5
enter second number7
12
<class 'int'>
```


Type Conversion

Implicit in Python:

- Implicit actions or conversions happen automatically without the need for explicit instructions.
- Implicit type conversion (coercion) occurs when Python automatically converts data types for operations.
- Python performs implicit actions to make code more readable and user-friendly.

Explicit in Python:

- Explicit actions or conversions require specific instructions provided by the programmer.
- Explicit type conversion (casting) is performed when you provide clear and direct commands for type conversion.
- Explicit actions are used when you need precise control and clarity in your code.

In [83]:
```
# Implicitly converts 'a' to a float before addition
a = 5
b = 2.0
result = a + b
```

In [84]:
```
# Explicitly convert the string to an integer
num_str = "42"
num_int = int(num_str)
num_int+b
```

Out[84]:
```
44.0
```

In [85]:
```
# Implicit Vs Explicit
print(5+5.6)
print(type(5),type(5.6))

print(4 + '4')
```

```
10.6
<class 'int'> <class 'float'>

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_7452\3295153562.py in <module>
      3 print(type(5),type(5.6))
      4
----> 5 print(4 + '4')

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```


In [86]:
```
# Explicit
# str -> int
#int(4+5j)

# int to str
str(5)

# float
float(4)
```

Out[86]:
```
4.0
```

7. Literals

In Python, literals are fixed values or data that are directly used in your code. They represent constants and can be assigned to variables

String Literals: These are sequences of characters enclosed in single (' '), double (" "), or triple ('' '' or """ """) quotes. For example:

In [94]:
```
print('1')
print('vishal')
print('2')
print("vishal")
print('3')
print('''vishal
hi''')
print('4')
print("""vishal
hi
b2
b7
d1""")
print('5')
print("vishal's")
print('vishal"s')
```

```
1
vishal
2
vishal
3
vishal
hi
4
vishal
hi
b2
b7
d1
5
vishal's
vishal"s
```

In [95]:
```
print('vishal's')
```

```
  File "C:\Users\VISHAL\AppData\Local\Temp\ipykernel_7452\927117232.py", l
ine 1
    print('vishal's')
                   ^
SyntaxError: invalid syntax
```

Numeric Literals: These are used to represent numeric values. They include integers, floating-point numbers, and complex numbers. For example:

In [2]:
```
int_literal = 42
float_literal = 3.14
complex_literal = 2 + 3j
print(int_literal)
print(float_literal)
print(complex_literal)
```

```
42
3.14
(2+3j)
```

Boolean Literals: These represent the two Boolean values, True and False

In [3]:
```
bool_literal_true = True
bool_literal_false = False
print(bool_literal_true)
print(bool_literal_false)
```

```
True
False
```

None Literal: The None literal represents the absence of a value or a null value

In [4]:
```
none_literal = None
print(none_literal)
```

```
None
```

List Literals: Lists are collections of values, and you can create them using square brackets.

In [5]:
```
list_literal = [1, 2, 3, 4]
print(list_literal)
```

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

Tuple Literals: Tuples are similar to lists but use parentheses for literals.

In [6]:
```
tuple_literal = (1, 2, 3, 4)
print(tuple_literal)
```

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

Dictionary Literals: Dictionaries are collections of key-value pairs, and you can create them using curly braces.

In [7]:
```
dict_literal = {"key1": "value1", "key2": "value2"}
print(dict_literal)
```

```
{'key1': 'value1', 'key2': 'value2'}
```

Set Literals: Sets are collections of unique elements and are created using curly braces with values separated by commas

In [8]:
```
set_literal = {1, 2, 3, 4}
print(set_literal)
```

```
{1, 2, 3, 4}
```

Raw String Literals: A raw string literal is prefixed with 'r' and is used to specify raw strings that don't escape backslashes.

In [109]:
```
raw_string_literal = r"C:\Users\Username"
print(raw_string_literal)
```

```
C:\Users\Username
```

In [110]:
```
raw_string_literal = "C:\Users\Username"
print(raw_string_literal)
```

```
  File "C:\Users\VISHAL\AppData\Local\Temp\ipykernel_7452\3293659636.py", line 1
    raw_string_literal = "C:\Users\Username"
                                            ^
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated \UXXXXXXXX escape
```

Formatted String Literals (f-strings): Introduced in Python 3.6, f-strings are used for string formatting by placing an 'f' or 'F' before the string literal.

In [112]:
```
name = "Alice"
formatted_string = f"Hello, {name}!"
print(formatted_string)
```

```
Hello, Alice!
```

In [113]:
```
#Complex Literal
x = 7.14j
print(x, x.imag, x.real)
```

```
7.14j 7.14 0.0
```

In [115]:
```
unicode = u"\U0001f600\U0001F606\U0001F923"
raw_str = r"raw \n string"
print(unicode)
```

```
😀😆🤣
```

In [116]:
```
a = True + 4
b = False + 10

print("a:", a)
print("b:", b)
```

```
a: 5
b: 10
```

In [118]:
```
k = None
a = 5
b = 6
print(a+b,k)
```

```
11 None
```

In [10]:
```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal
print(a,b,c,d)
```

```
10 100 200 300
```


None is a special built-in constant that represents the absence of a value or a null value. It is often used to signify that a variable or object has no assigned value.

- When you set a variable to None, you are essentially saying that the variable exists, but it doesn't contain any meaningful data. This can be useful in various situations, such as when you want to initialize a variable before assigning a real value to it

Operators in Python

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Membership Operators

Arithmetic Operators

+ (Addition): This operator is used to add two numbers.

In [11]:
```
result = 5 + 3  # result will be 8
print(result)
```

```
8
```

- (Subtraction): Subtracts the right operand from the left operand.

In [12]:
```
result = 10 - 3  # result is 7
print(result)
```

```
7
```

* (Multiplication): Multiplies two numbers

In [13]:
```
result = 4 * 6  # result is 24
print(result)
```

```
24
```


/ (Division): Divides the left operand by the right operand (float division).

In [14]:
```
result = 20 / 4  # result is 5.0
print(result)
```

```
5.0
```

// (Floor Division): Divides and rounds down the result to the nearest whole number.

In [15]:
```
result = 20 // 4  # result is 5
print(result)
```

```
5
```

% (Modulus): Divides and returns the remainder.

In [16]:
```
result = 20 % 3  # result is 2 (20 divided by 3 leaves a remainder of 2
print(result)
```

```
2
```

** (Exponentiation): Raises the left operand to the power of the right operand.

In [17]:
```
result = 2 ** 3  # result is 8 (2 raised to the power of 3)
print(result)
```

```
8
```

Comparison operators

- used to compare values and return either True or False based on the comparison.
- They play a crucial role in control structures like if statements and loops, as well as in various conditional expressions and algorithms.

== (Equal): Compares whether two values are equal. Returns True if they are equal and False otherwise.

In [18]:
```
result=5==5
print(result)
```

```
True
```

In [19]:
```
4.0==4
```

Out[19]: True

In [20]:
```
4=="4"
```

Out[20]: False

In [21]:
```
True==1.0
```

Out[21]: True

In [22]:
```
False==0.0
```

Out[22]: True

In [23]:
```
True==1
```

Out[23]: True

!= (Not Equal): Checks if two values are not equal. Returns True if they are different and False if they are equal.

In [24]:
```
result=5!=5
print(result)
```

```
False
```

In [25]:
```
5.0!=5
```

Out[25]: False

In [26]:
```
5.0!="5"
```

Out[26]: True

< (Less Than): Determines if the left operand is less than the right operand. Returns True if it's true and False otherwise.

In [27]:
```
result=5<4
print(result)
```

```
False
```

In [28]:
```
5<True
```

Out[28]: False

In [29]:
```
False<True
```

Out[29]: True

> (Greater Than): Checks if the left operand is greater than the right operand. Returns True if it's true and False otherwise.

In [30]:
```
result=5>4
print(result)
```

```
True
```

In [31]:
```
5>True
```

Out[31]: True

In [32]:
```
False>True
```

Out[32]: False

In [33]:
```
1>True
```

Out[33]: False

<= (Less Than or Equal To): Verifies if the left operand is less than or equal to the right operand. Returns True if it's true and False otherwise.

In [34]:
```
result=5<=4
print(result)
```

```
False
```

In [35]:
```
5<=5
```

Out[35]: True

>= (Greater Than or Equal To): Determines if the left operand is greater than or equal to the right operand. Returns True if it's true and False otherwise.

In [36]:
```
result=5>=4
print(result)
```

```
True
```

In [37]:
```
5>=4
```

Out[37]:
```
True
```

Logical operators

- in Python allow you to perform logical operations on boolean values (either True or False). They are often used to combine or manipulate boolean values to make decisions in your code.

and (Logical AND): The and operator returns True if both operands are True. If at least one operand is False, it returns False. It can be used to check if multiple conditions are met.

In [38]:
```
a = 5 and 8
print(a)
```

```
8
```

In [39]:
```
a=5
if a==5 and a!=6:
    print(a)
```

```
5
```

In [40]:
```
a=5
if a==5 and a==6:
    print(a)
else:
    print('hi')
```

```
hi
```

In [41]:
```
is_sunny = True
is_warm = True
if is_sunny and is_warm:
    print("It's a sunny and warm day.")
```

```
It's a sunny and warm day.
```

In [42]:
```
is_raining = True
is_cold = False
if is_raining and is_cold:
    print("It's raining and cold.")
```

In [43]:
```
a= 1 and False
print(a)
```

```
False
```

In [44]:
```
a= 0 and True
print(a)
```

```
0
```


or (Logical OR): The or operator returns True if at least one of the operands is True. It returns False only if both operands are False. It's useful for situations where you want to check if at least one condition is met.

In [45]:
```
has_ticket = True
has_id = False
if has_ticket or has_id:
    print("You can enter the event.")

```

```
You can enter the event.
```

In [46]:
```
a= 5 or 6
print(a)
```

```
5
```

In [47]:
```
a= 0 or 8
print(a)
```

```
8
```

In [48]:
```
a= 8 or 0
print(a)
```

```
8
```

not (Logical NOT): The not operator is a unary operator that negates the boolean value of its operand. It returns True if the operand is False, and False if the operand is True. It's used to reverse the boolean value.

In [49]:
```
a=True
print(not a)
```

```
False
```

In [50]:
```
a=0
print(not a)
```

```
True
```


Assignment operators

- used to assign values to variables and, in some cases, update the value of a variable while performing an operation.

= (Assignment): The = operator assigns the value on the right side to the variable on the left side.

In [51]:
```
x = 5  # Assigns the value 5 to the variable x
print(x)
```

```
5
```

+= (Add and Assign): The += operator adds the value on the right side to the variable on the left side and assigns the result to the variable on the left

In [52]:
```
y = 10
y += 3  # Equivalent to y = y + 3
# y now has the value 13
print(y)
```

```
13
```

-= (Subtract and Assign): The -= operator subtracts the value on the right side from the variable on the left side and assigns the result to the variable on the left.

In [53]:
```
z = 15
z -= 6  # Equivalent to z = z - 6
# z now has the value 9
print(z)
```

```
9
```

***= (Multiply and Assign): The *= operator multiplies the variable on the left side by the value on the right side and assigns the result to the variable on the left.**

In [54]:
```
a = 4
a *= 7  # Equivalent to a = a * 7
# a now has the value 28
print(a)
```

```
28
```

/= (Divide and Assign): The /= operator divides the variable on the left side by the value on the right side and assigns the result to the variable on the left.

In [55]:
```
b = 30
b /= 3  # Equivalent to b = b / 3
# b now has the value 10.0 (note the float division)
print(b)
```

```
10.0
```

//= (Floor Divide and Assign): The //= operator performs floor division on the variable on the left side by the value on the right side and assigns the result to the variable on the left.

In [56]:
```
c = 17
c //= 5  # Equivalent to c = c // 5
# c now has the value 3
print(c)
```

```
3
```

%= (Modulus and Assign): The %= operator calculates the remainder when dividing the variable on the left side by the value on the right side and assigns the remainder to the variable on the left.

In [57]:
```
d = 25
d %= 7  # Equivalent to d = d % 7
# d now has the value 4
print(d)
```

```
4
```

**= (Exponentiation and Assign): The **= operator raises the variable on the left side to the power of the value on the right side and assigns the result to the variable on the left.

In [58]:
```
e = 2
e **= 3  # Equivalent to e = e ** 3
# e now has the value 8
print(e)
```

```
8
```


Bitwise operators

- used to perform operations on individual bits (0s and 1s) of integer values. They are more common in low-level programming, such as embedded systems and systems programming

```
# decimal integer 27 to binary:

Start with 27.
Divide 27 by 2: Quotient = 13, Remainder = 1.
Write down the remainder as the rightmost digit: 1.
Set the quotient to 13.
Repeat:
Divide 13 by 2: Quotient = 6, Remainder = 1.
Write down the remainder: 11.
Set the quotient to 6.
Divide 6 by 2: Quotient = 3, Remainder = 0.
Write down the remainder: 011.
Set the quotient to 3.
Divide 3 by 2: Quotient = 1, Remainder = 1.
Write down the remainder: 1011.
Set the quotient to 1.
Divide 1 by 2: Quotient = 0, Remainder = 1.
Write down the remainder: 11011.
The binary representation of 27 is 11011.
So, the decimal integer 27 is equivalent to the binary number 11011.

# Start with the binary number you want to convert.
Examine each digit in the binary number from right to left.
```

```
For each digit, multiply it by 2 raised to the power of its position (starting with 0 for the rightmost digit).
Sum the results of these multiplications.
Here's a step-by-step example of converting the binary number 11011 to an integer:

Start with the binary number: 11011.

Examine each digit from right to left:

The rightmost digit is 1, so it's multiplied by 2^0 (which is 1).
The next digit is 1, so it's multiplied by 2^1 (which is 2).
The next digit is 0, so it's multiplied by 2^2 (which is 4).
The next digit is 1, so it's multiplied by 2^3 (which is 8).
The leftmost digit is 1, so it's multiplied by 2^4 (which is 16).
Sum the results:

1*1 + 1*2 + 0*4 + 1*8 + 1*16 = 1 + 2 + 0 + 8 + 16 = 27
So, the binary number 11011 is equivalent to the decimal integer 27.
```


& (Bitwise AND): The & operator performs a bitwise AND operation between two integers. It returns a new integer with 1s in positions where both operands have 1s; otherwise, it sets the bit to 0.

In [194]:

```python
x = 5  # Binary: 0101
y = 3  # Binary: 0011
result = x & y  # Binary result: 0001 (Decimal result: 1)
print(result)
```

```
1
```

| (Bitwise OR): The | operator performs a bitwise OR operation between two integers. It returns a new integer with 1s in positions where at least one operand has a 1.

In [195]:

```python
a = 5  # Binary: 0101
b = 3  # Binary: 0011
result = a | b  # Binary result: 0111 (Decimal result: 7)
print(result)
```

```
7
```

^ (Bitwise XOR): The ^ operator performs a bitwise XOR (exclusive OR) operation between two integers. It returns a new integer with 1s in positions where only one operand has a 1

In [196]:
```
p = 5  # Binary: 0101
q = 3  # Binary: 0011
result = p ^ q  # Binary result: 0110 (Decimal result: 6)
print(result)
```

```
6
```

~ (Bitwise NOT): The ~ operator performs a bitwise NOT operation on a single integer. It flips all the bits, turning 1s into 0s and vice versa. Be cautious with this operator because it also inverts the sign of the number

In [201]:
```
r = 5  # Binary: 0000 0101
result = ~r  # Binary result: 1111 1010 (Decimal result: -6)
print(result)
```

```
-6
```

<< (Left Shift): The << operator shifts the bits of an integer to the left by a specified number of positions. It effectively multiplies the number by 2 raised to the power of the shift count.

In [192]:
```
s = 5  # Binary: 0000 0101
result = s << 2  # Binary result: 0001 0100 (Decimal result: 20)
print(result)
```

```
20
```

>> (Right Shift): The >> operator shifts the bits of an integer to the right by a specified number of positions. It effectively performs integer division by 2 raised to the power of the shift count.

In [199]:
```
t = 20  # Binary: 0001 0100
result = t >> 2  # Binary result: 0000 0101 (Decimal result: 5)
print(result)
```

```
5
```

Membership operators

- used to check whether a specific value is a member of a sequence or collection, such as a list, tuple, string, or set. There are two membership operators: in and not in

in (Membership Operator): The in operator checks if a value exists in a given sequence or collection. If the value is found in the sequence, it returns True; otherwise, it returns False.

In [202]:
```
my_list = [1, 2, 3, 4, 5]
result = 3 in my_list  # True, because 3 is in the list
print(result)
```

```
True
```

not in (Membership Operator): The not in operator checks if a value is not found in a given sequence. If the value is not in the sequence, it returns True; otherwise, it returns False.

In [203]:
```
# True, because "Goodbye" is not in the string
my_string = "Hello, World!"
result = "Goodbye" not in my_string
```

Identity operators in Python are used to compare the memory locations (identities) of objects rather than their values.

is (Identity Operator): The is operator checks if two objects are the same, meaning they share the same memory location. If the objects have the same identity, it returns True; otherwise, it returns False

In [205]:
```
x = [1, 2, 3]
y = x  # y refers to the same object as x
result = x is y  # True, because x and y are the same object
print(result)
```

```
True
```

is not (Identity Operator): The is not operator checks if two objects are not the same. If the objects do not have the same identity, it returns True; otherwise, it returns False.

In [206]:
```
a = [1, 2, 3]
b = [1, 2, 3]  # b is a different object with the same value
result = a is not b  # True, because a and b are different objects
print(result)
```

```
True
```

The Ternary Conditional Operator in Python is a shorthand way to express conditional

statements, allowing you to write a simple conditional expression in a single line. It's also known as the conditional expression. The general syntax of the ternary conditional operator is as follows:

- value_if_true if condition else value_if_false
- condition is a boolean expression that is evaluated.
- If the condition is True, the expression returns value_if_true.
- If the condition is False, the expression returns value_if_false

In [208]:
```
age = 18
status = "Adult" if age >= 18 else "Minor"
print(status)
```

```
Adult
```

Precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated first. If two operators have the same precedence, then associativity determines the order of evaluation. Operators with left-to-right associativity are evaluated from left to right, while operators with right-to-left associativity are evaluated from right to left

Operator	Description	Associativity
( )	Parentheses	left to right
**	Exponent	right to left
* / %	Multiplication / division / modulus	left to right
+ -	Addition / subtraction	left to right
<< >>	Bitwise left shift / Bitwise right shift	left to right
< <= > >=	Relational operators: less than / less than or equal to / greater than / greater than or equal to	left to right
== !=	Relational operators: is equal to / is not equal to	left to right
is, is not in, not in	Identity operators Membership operators	left to right
&	Bitwise AND operator	left to right
^	Bitwise exclusive OR operator	left to right
|	Bitwise inclusive OR operator	left to right
not	Logical NOT	right to left
and	Logical AND	left to right
or	Logical OR	left to right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operators: Addition / subtraction Multiplication / division Modulus / bitwise AND Bitwise exclusive / inclusive OR Bitwise shift left / right shift	right to left

In [211]:
```
result = 5 + 3 * 2**3**2
print(result)
```

```
1541
```

In [214]:
```
# Result will be True (relational operators are evaluated left to right
x = 10
y = 15
result = x < y or x == y
```

In [216]:
```
a = True
b = False
c = True
result = a and b or c
# Result will be True (logical operators have precedence)
print(result)
```

```
True
```

In [7]:
```
result = 2**3 + 10 / 2 - 1 < 5 and (
    7 or 3) != 6 or "Python" in ["Java", "Python", "C++"]
print(result)
```

```
True
```

In [4]:
```
result = (8 % 3) ** 2 +(1 > 10 or
(not True and (3 in [1, 2, 3])) or "hello" != "world" or (5 // 2) + 1 =
print(result)
```

```
5
```

In [1]:
```
5 or 1/0
```

Out[1]:
```
5
```

In [2]:
```
1/0 or 6
```

```
---------------------------------------------------------------------------
-
ZeroDivisionError                         Traceback (most recent call las
t)
~\AppData\Local\Temp\ipykernel_7964\2162312769.py in <module>
----> 1 1/0 or 6

ZeroDivisionError: division by zero
```

In [ ]:
```

```