Python Refresher

Agenda

- What is Python?
- Data Types
- Variables
- Keywords
- Inputs
- Operators
- Type Casting

What is Python?

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python's design philosophy emphasizes code readability and simplicity, making it an ideal choice for both beginners and experienced programmers. Here's a detailed definition:

Python Programming Language

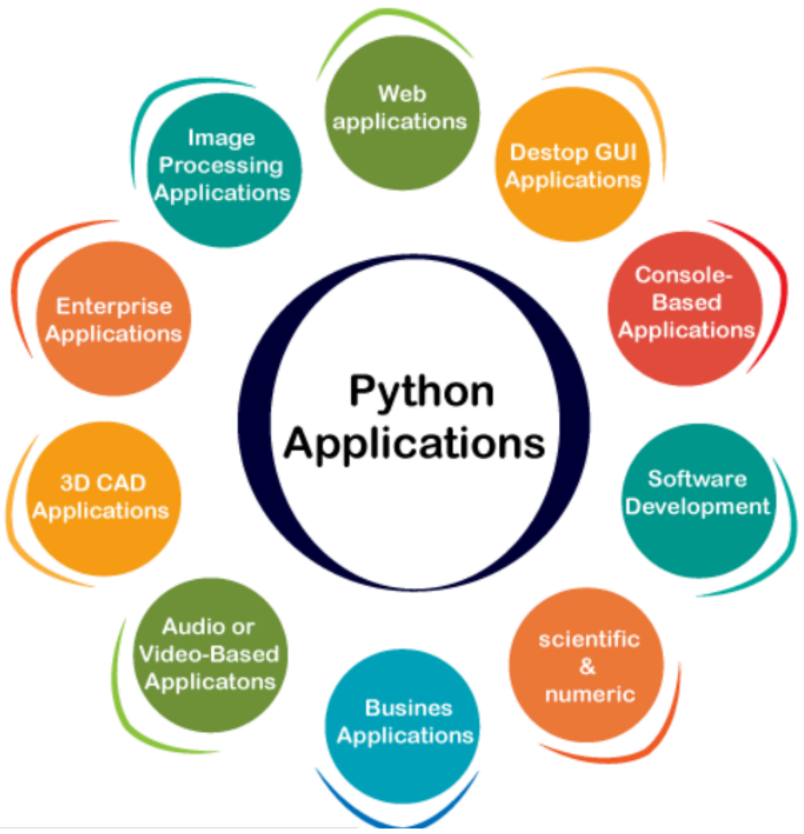
1. **High-Level Language:**

   Python is a high-level language, meaning it abstracts away most of the complex details of the computer's hardware. This allows developers to focus on writing logic rather than managing memory and other lower-level tasks.

2. **Interpreted Language:**

   Unlike compiled languages (like C or C++), Python is interpreted. This means that Python code is executed line-by-line, which facilitates a rapid development cycle. However, it may also result in slower execution speeds compared to compiled languages.

Applications of python



- **Web Applications:** Python's frameworks like Django and Flask are used to build dynamic and scalable web applications.
- **Desktop GUI Applications:** Python libraries such as Tkinter and PyQt enable the creation of user-friendly desktop applications.
- **Console-Based Applications:** Python excels at creating command-line interfaces with libraries like Argparse and the standard IO modules.
- **Software Development:** Python is used for building control, testing, and project management in the software development process.
- **Scientific & Numeric:** Python's libraries like SciPy and NumPy support complex mathematical and scientific computations.
- **Business Applications:** Python's readability and scalability make it ideal for developing ERP and e-commerce business applications.
- **Audio or Video-Based Applications:** Python's multimedia libraries such as Pyglet and GStreamer are used to develop audio and video applications.
- **3D CAD Applications:** Python can create and manipulate 3D models with libraries like Fandango and HeeksCNC.
- **Enterprise Applications:** Python is capable of developing scalable applications for enterprise-level needs, such as OpenERP and Tryton.
- **Image Processing Applications:** Python provides powerful libraries like OpenCV and Pillow for image processing and manipulation.

Data Type

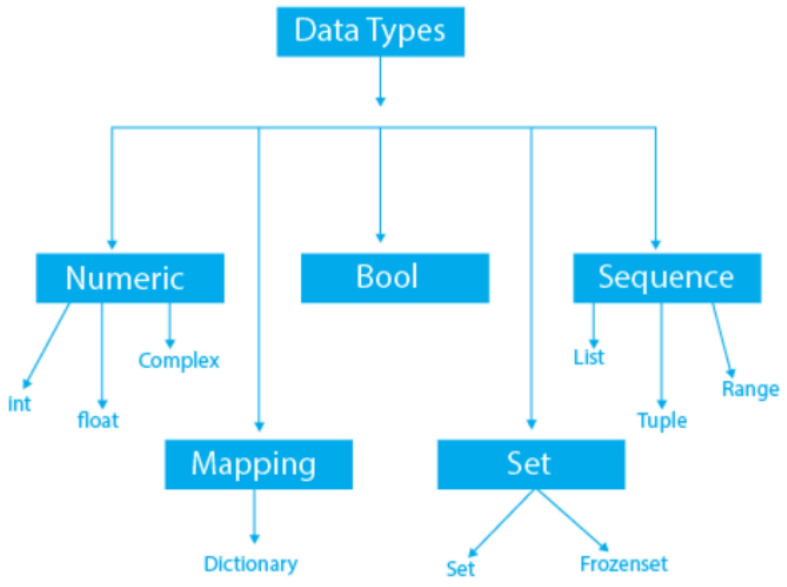
A data type is a category that defines what kind of value a variable can store and how it can be used in a program. Data types determine the possible operations on the data and how it is stored in memory. Different programming languages have different data types, which describe the nature of the data a variable can hold.

Key Points:

- **Purpose:**
    - To define the nature of data that can be stored in a variable.
    - To determine the operations that can be performed on the data.
    - To manage the memory allocation for storing the data.
- **Why Data Types Matter:**
    - **Memory Management:** Different data types require different amounts of memory. For example, an integer typically requires less memory than a string.
    - **Operations:** Certain operations are only valid for specific data types. For instance, you can add two integers but cannot directly add an integer and a string without converting them.
    - **Error Prevention:** Properly using data types helps prevent errors and bugs in the code. Type checking can ensure that operations are performed on compatible data types.

Types of Data type

- **Primitive data types -** Primitive data types, also known as basic or fundamental data types, are the most basic types of data that are built into a programming language. They serve as the building blocks for data manipulation and are typically defined at the language level. Primitive data types are not composed of other data types and generally represent single values.
- **Container -** In Python, container data types are types that can hold multiple items or elements. These items can be of any data type, including other containers. Containers provide ways to group related data and manage collections of items efficiently



Numeric

- **Integer (int)**

  Definition: Integers are whole numbers without a fractional component.

  Example: x = 5

```
a = 10
type(a)
```

```
int
```

- **Floating Point (float)**

  Definition: Floating point numbers are real numbers with a fractional component.

  Example: y = 3.14

```
a = 10.5
type(a)
```

```
float
```

- **Complex Number (complex)**

  Definition: Complex numbers have a real part and an imaginary part, denoted as a + bj, where a is the real part and b is the imaginary part.

  Example: z = 1 + 2j

```
z = 1 + 2j
type(z)
```

```
complex
```

Bool

- **Boolean (bool)**

  Definition: Booleans represent one of two values: True or False.

  Example: is_valid = True

```
is_valid = True
type(is_valid)
```

```
bool
```

Sequence

- **String (str)**

  Definition: Strings are sequences of characters, enclosed in single quotes ('...') or double quotes ("...").

  Example: name = "Alice"

```
name = "alice"
type(name)
```

```
str
```

- **List (list)**

  Definition: Lists are ordered, mutable sequences of elements that can be of different types.

  Example: numbers = [1, 2, 3, 4, 5]

```
numbers = [1, 2, 3, 4, 5]
type(numbers)
```

```
list
```

- **Tuple (tuple)**

  Definition: Tuples are ordered, immutable sequences of elements that can be of different types.

  Example: coordinates = (10.0, 20.0)

```
coordinates = (10.0, 20.0)
type(coordinates)
```

```
tuple
```

Set

- **Set (set)**

  Definition: Sets are unordered collections of unique elements.

  Example: unique_numbers = {1, 2, 3, 4, 5}

```
unique_numbers = {1, 2, 3, 4, 5}
type(unique_numbers)
```

```
set
```

- **Frozenset (frozenset)**

  Definition: Frozensets are immutable sets, used for creating sets that cannot be modified after creation.

  Example: immutable_set = frozenset([1, 2, 3])

```
immutable_set = frozenset([1, 2, 3])
type(immutable_set)
```

```
frozenset
```

Mapping

- **Dictionary (dict)**

Definition: Dictionaries are unordered collections of key-value pairs, where keys are unique and can be of different types.

Example: person = {"name": "Alice", "age": 25}

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

```
dict
```

Bytes

- **Bytes (bytes)**

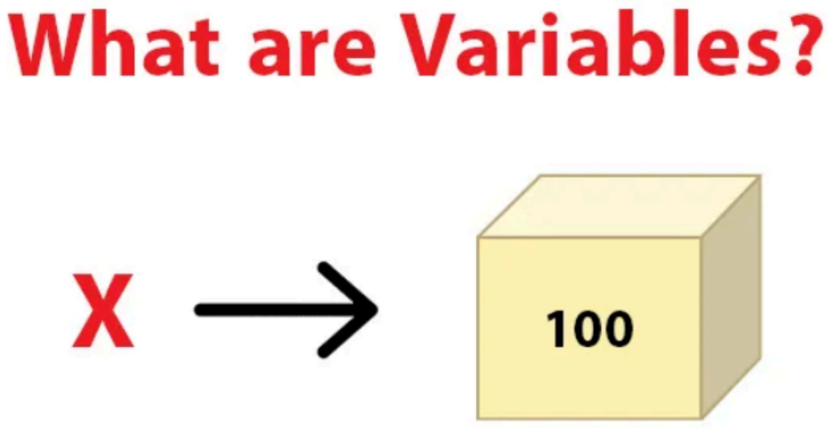
  Definition: Bytes are immutable sequences of bytes, used for binary data.

  Example: data = b'hello'

```
data = b'hello'
type(data)
```

```
bytes
```

Variable



A variable is a fundamental concept in programming and computer science that represents a storage location in memory, identified by a name (an identifier), which holds a value that can be modified during the execution of a program. Variables are used to store data that can be referenced and manipulated by the code. They allow programmers to write flexible and dynamic programs by providing a way to label and keep track of information. Variables have a data type, which defines the kind of data they can store, such as integers, floats, strings, or objects.

```
# Variable assignment
age = 25          # 'age' is a variable storing an integer value
name = "Alice"    # 'name' is a variable storing a string value
height = 5.7      # 'height' is a variable storing a float value

# Using variables
print(name + " is " + str(age) + " years old and " + str(height) + " feet tall.")
```

```
Alice is 25 years old and 5.7 feet tall.
```

Rules for Naming Variables

- **Start with a letter or an underscore:** Variable names must begin with a letter (a-z, A-Z) or an underscore (_).
- **Follow with letters, digits, or underscores:** The rest of the name can include letters, digits (0-9), and underscores.
- **Case-sensitive:** Variable names are case-sensitive (e.g., Rahul and rahul are different variables).
- **No special characters:** Variable names cannot contain special characters (e.g., !, @, #, %).

- **Avoid keywords:** Variable names should not be the same as any Python keyword.

Variable Naming Conventions

- **Camel Case:** myVariableName
- **Pascal Case:** MyVariableName
- **Snake Case:** my_variable_name

Example of Valid Identifiers

```
a123 = 10
_n = "hello"
n_9 = 5.7
```

Example of Invalid Identifiers

```
1a = 10  # Starts with a digit
n%4 = 5  # Contains a special character
n 9 = 8  # Contains a space
```

```
  File "<ipython-input-23-9aa497d8a70b>", line 1
    1a = 10  # Starts with a digit
    ^
SyntaxError: invalid decimal literal
```

Types of Variable

Variable Types There are two types of variables in Python: local and global.

- **Local Variables:** Declared inside a function and can only be accessed within that function.

```
def add():
    a = 20
    b = 30
    c = a + b
    print("The sum is:", c)

add()  # Output: The sum is: 50
```

```
The sum is: 50
```

- **Global Variables:** Declared outside any function and can be accessed globally throughout the program.

```
x = 101

def mainFunction():
    global x
    print(x)
    x = 'Welcome To Python'
    print(x)

mainFunction()  # Output: 101, Welcome To Python
print(x)  # Output: Welcome To Python
```

```
101
Welcome To Python
Welcome To Python
```

Deleting Variables

You can delete a variable using the "del" keyword.

```
x = 6
print(x)  # Output: 6
```

```
6
```

```
del x
print(x)  # Output: NameError: name 'x' is not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-b9b42d3a78e3> in <cell line: 2>()
      1 del x
----> 2 print(x)  # Output: NameError: name 'x' is not defined

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

Multiple Assignment

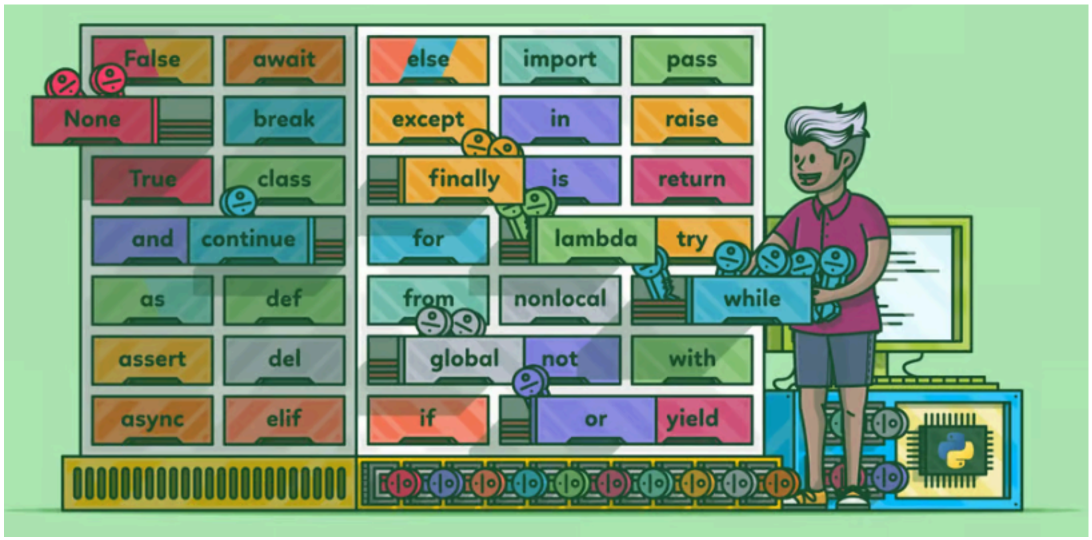
You can assign values to multiple variables in one statement.

```
x = y = z = 50
print(x, y, z)  # Output: 50 50 50

a, b, c = 5, 10, 15
print(a, b, c)  # Output: 5 10 15
```

```
50 50 50
5 10 15
```

Keywords



Python keywords are special reserved words that have specific meanings and purposes and can't be used for anything but those specific purposes. These keywords are always available—you'll never have to import them into your code.

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

```
>>> help("keywords")
```

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

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

Python also provides a keyword module for working with Python keywords in a programmatic way. The keyword module in Python provides two helpful members for dealing with keywords:

kwlist provides a list of all the Python keywords for the version of Python you're running. iskeyword() provides a handy way to determine if a string is also a keyword.

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'async']
>>> len(keyword.kwlist)
```

```
35
```

Key characteristics of keywords:

- **Predefined and reserved:** These words have fixed meanings assigned to them by the language itself. You cannot alter their definition or use them for other purposes in your code.
- **Always available:** Keywords are built-in to Python, so you don't need to import any modules to use them.
- **Case-sensitive:** Python treats keywords as case-sensitive. For instance, if is a keyword, but If is not.

Python keywords are essential for writing clean, efficient, and effective code. They help define the structure and flow of the program and facilitate various functionalities such as control flow, error handling, and defining functions and classes. Understanding and using these keywords correctly is fundamental to becoming proficient in Python programming.

Inputs

TOPIC

input() Function in Python

How to take input in python

In Python, taking input from the user is typically done using the input() function. This function reads a line from the input (usually from the keyboard), converts it to a string, and returns it. If you need the input to be of a different type (such as an integer or float), you'll need to convert the string using the appropriate type conversion functions.

Here's a detailed explanation of how to take input in Python:

Basic Input To take a simple input from the user, use the input() function.

```
name = input("Enter your name: ")
print("Hello, " + name)
```

```
Enter your name: vaibhav
Hello, vaibhav
```

Converting Input to Other Types

By default, the input() function returns a string. To convert the input to other data types, you can use type conversion functions.

```
age = int(input("Enter your age: "))
print("You are", age, "years old")
```

```
Enter your age: 13
You are 13 years old
```

Float Input

```
height = float(input("Enter your height in meters: "))
print("Your height is", height, "meters")
```

```
Enter your height in meters: 10
Your height is 10.0 meters
```

List Input

To take a list as input, you can read the input as a string and then split it into a list using the split() method.

```
numbers = input("Enter numbers separated by spaces: ").split()
numbers = [int(num) for num in numbers]
print("You entered the numbers:", numbers)
```

```
Enter numbers separated by spaces:  3 4 5 6 7 8 9
You entered the numbers: [3, 4, 5, 6, 7, 8, 9]
```

Taking Multiple Inputs

You can take multiple inputs in a single line by using the split() method.

```
a, b = input("Enter two numbers separated by space: ").split()
a = int(a)
b = int(b)
print("The first number is", a)
print("The second number is", b)
```

```
Enter two numbers separated by space: 6 10
The first number is 6
The second number is 10
```

Using map() for Multiple Inputs

The map() function can be used to apply a function to all items in an input list.

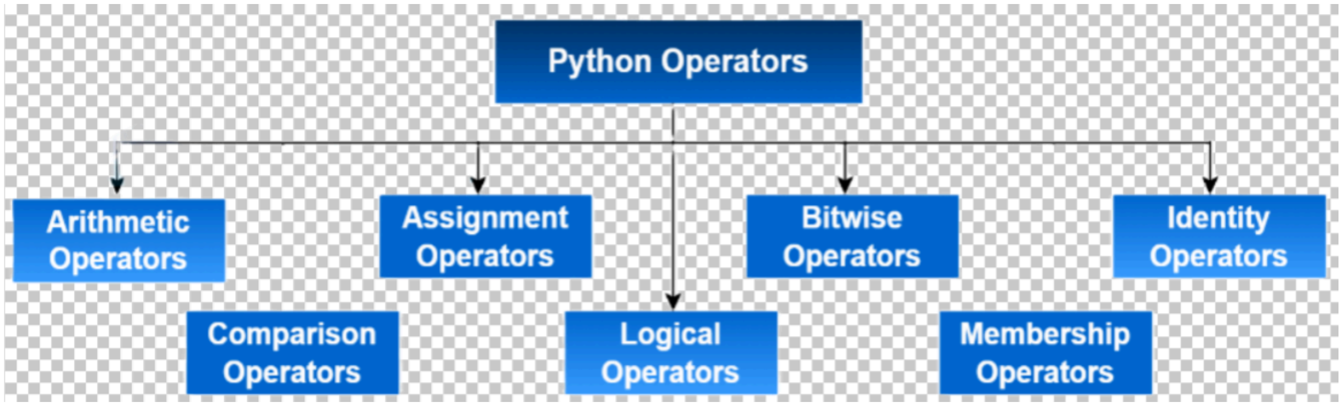
```
a, b = map(int, input("Enter two numbers separated by space: ").split())
print("The first number is", a)
print("The second number is", b)
```

```
Enter two numbers separated by space: 4 10
The first number is 4
The second number is 10
```

Taking input in Python is straightforward with the **input()** function, and you can convert the input to different types as needed. Handling user input carefully with appropriate type conversion and error handling ensures robust and user-friendly programs.

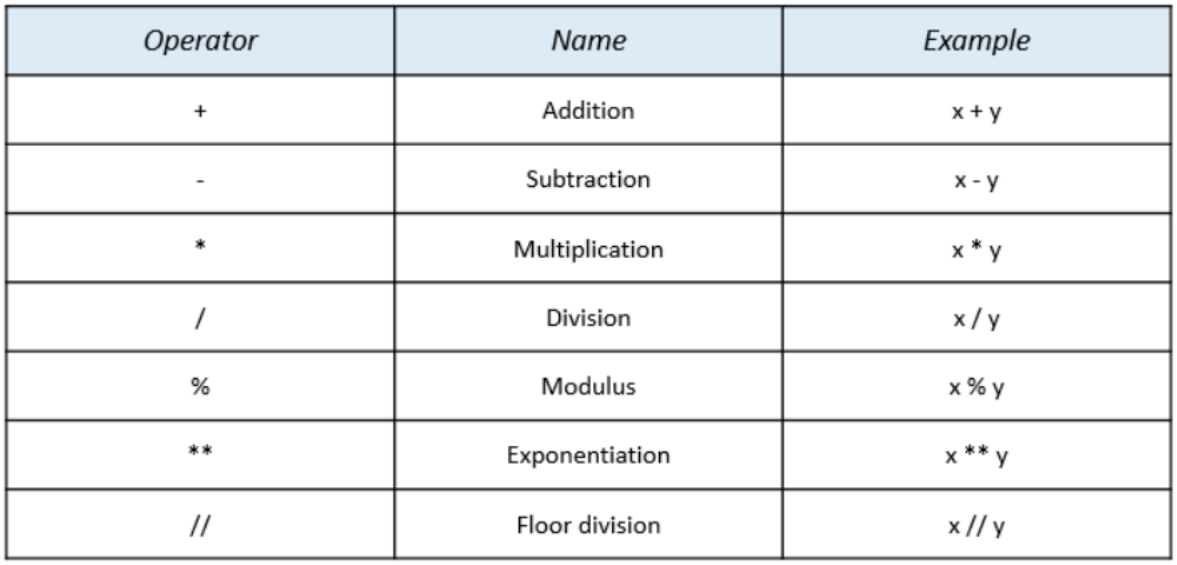
Operators

Operators are special symbols or keywords that carry out arithmetic or logical computation. The value that the operator operates on is called the operand. Python supports various types of operators, which can be classified into several categories:



1. Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.



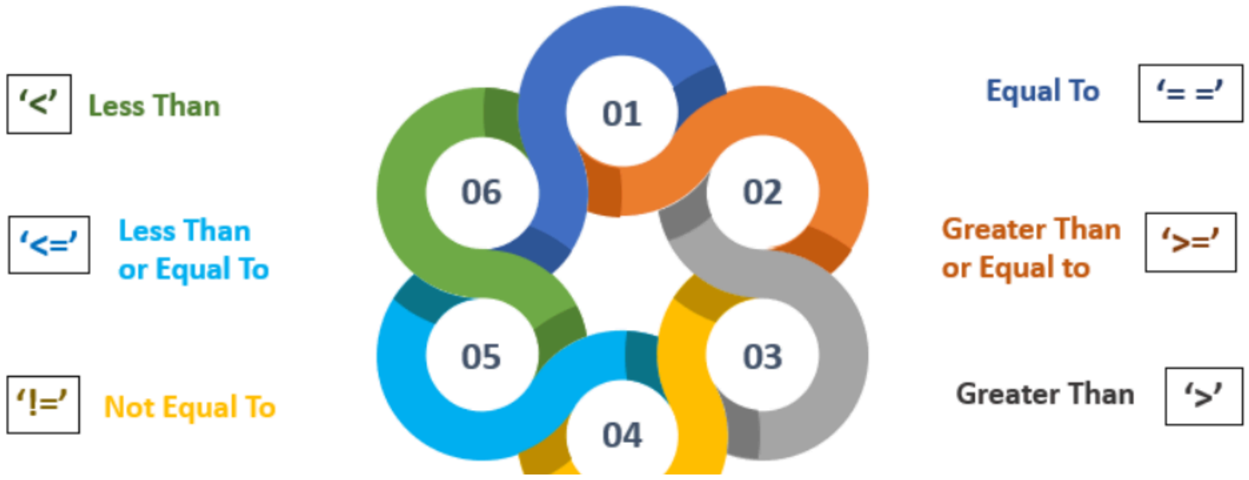
Operator	*Name*	*Example*
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

```
a = 10
b = 3
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
print("Floor Division:", a // b)
```

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Modulus: 1
Exponentiation: 1000
Floor Division: 3
```

2. Comparison Operators

Comparison operators compare the values of two operands and return a Boolean value.

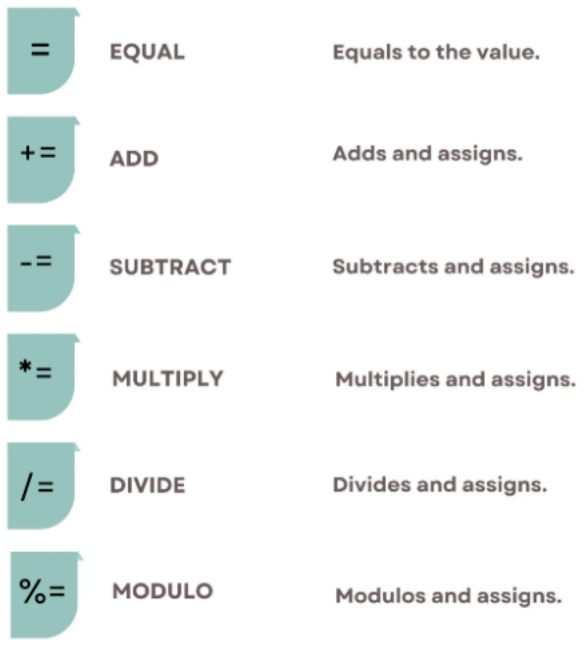


```
print("Equal:", a == b)
print("Not equal:", a != b)
print("Greater than:", a > b)
print("Less than:", a < b)
print("Greater than or equal to:", a >= b)
print("Less than or equal to:", a <= b)
```

```
Equal: False
Not equal: True
Greater than: True
Less than: False
Greater than or equal to: True
Less than or equal to: False
```

3. Assignment Operators

Assignment operators are used to assign values to variables.



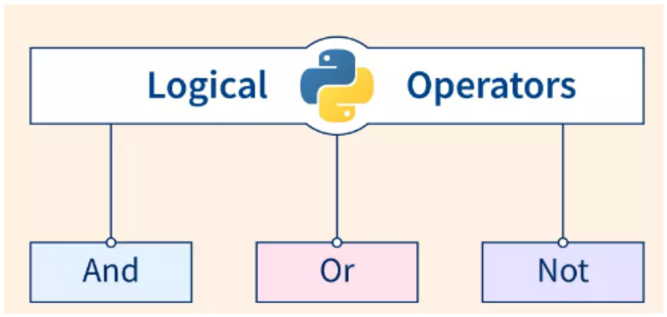
=	EQUAL	Equals to the value.
+=	ADD	Adds and assigns.
-=	SUBTRACT	Subtracts and assigns.
*=	MULTIPLY	Multiplies and assigns.
/=	DIVIDE	Divides and assigns.
%=	MODULO	Modulos and assigns.

```
c = 5
c += 2
print("Add and assign:", c)
c -= 2
print("Subtract and assign:", c)
c *= 2
print("Multiply and assign:", c)
c /= 2
print("Divide and assign:", c)
c %= 2
print("Modulus and assign:", c)
c **= 2
print("Exponent and assign:", c)
c //= 2
print("Floor divide and assign:", c)
```

```
Add and assign: 7
Subtract and assign: 5
Multiply and assign: 10
Divide and assign: 5.0
Modulus and assign: 1.0
Exponent and assign: 1.0
Floor divide and assign: 0.0
```

4. Logical Operators

Logical operators are used to perform logical operations and return a Boolean result.

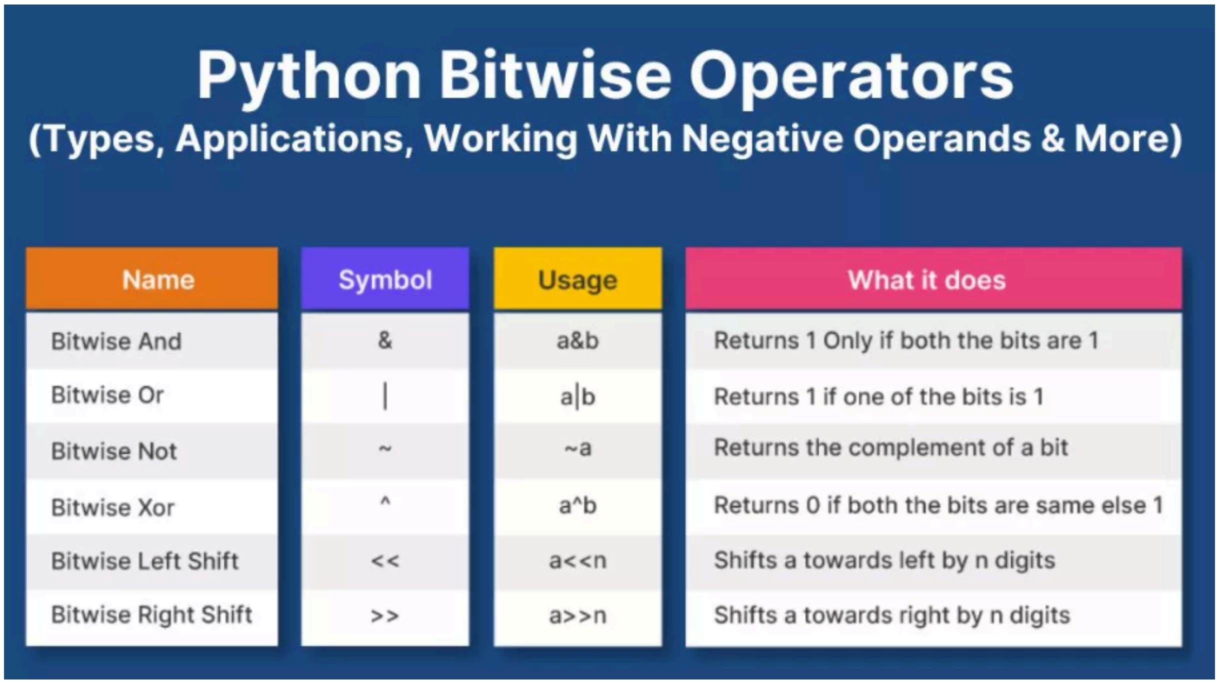


```
x = True
y = False
print("Logical AND:", x and y)
print("Logical OR:", x or y)
print("Logical NOT:", not x)
```

```
Logical AND: False
Logical OR: True
Logical NOT: False
```

5. Bitwise Operators

Bitwise operators are used to perform bit-level operations on integer types.



Python Bitwise Operators
(Types, Applications, Working With Negative Operands & More)

Name	Symbol	Usage	What it does
Bitwise And	&	a&b	Returns 1 Only if both the bits are 1
Bitwise Or	|	a|b	Returns 1 if one of the bits is 1
Bitwise Not	~	~a	Returns the complement of a bit
Bitwise Xor	^	a^b	Returns 0 if both the bits are same else 1
Bitwise Left Shift	<<	a<<n	Shifts a towards left by n digits
Bitwise Right Shift	>>	a>>n	Shifts a towards right by n digits

```
d = 2
e = 3
print("Bitwise AND:", d & e)
print("Bitwise OR:", d | e)
print("Bitwise XOR:", d ^ e)
print("Bitwise NOT:", ~d)
print("Bitwise left shift:", d << 1)
print("Bitwise right shift:", d >> 1)
```

```
Bitwise AND: 2
Bitwise OR: 3
Bitwise XOR: 1
Bitwise NOT: -3
Bitwise left shift: 4
Bitwise right shift: 1
```

6. Membership Operator

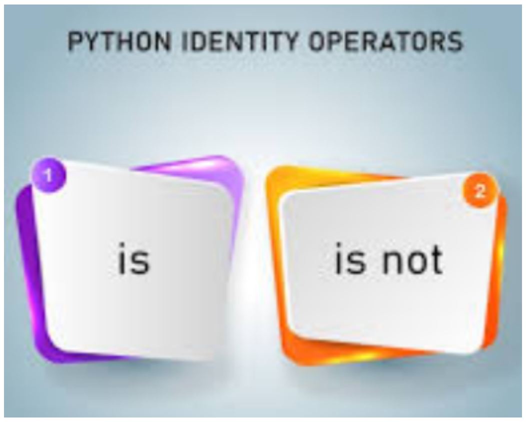
Membership operators are used to test if a sequence is present in an object.

```
my_list = [1, 2, 3, 4, 5]
print("Is 3 in list:", 3 in my_list)
print("Is 6 not in list:", 6 not in my_list)
```

```
Is 3 in list: True
Is 6 not in list: True
```

7. Identity Operators

Identity operators are used to compare the memory locations of two objects.



```
f = 5
g = 5
print("f is g:", f is g)
print("f is not g:", f is not g)
```

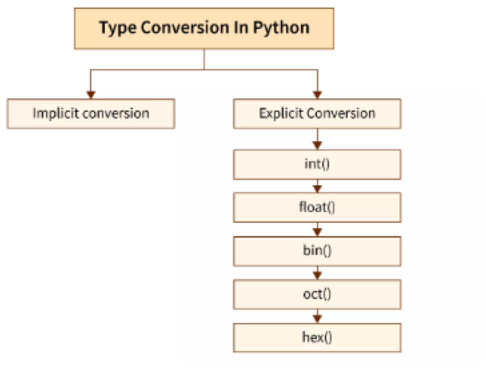
```
f is g: True
f is not g: False
```

Typecasting

Typecasting in Python, also known as type conversion, refers to converting one data type into another. Python provides several built-in functions to facilitate typecasting.

Types of Typecasting

1. Implicit Typecasting
2. Explicit Typecasting



Implicit Typecasting

Implicit typecasting occurs automatically when Python converts one data type to another without the need for explicit intervention by the user. This usually happens when performing operations between different data types.

```
x = 5    # Integer
y = 3.0  # Float

result = x + y  # x is implicitly converted to float
print(result)   # Output: 8.0
print(type(result))  # Output: <class 'float'>
```

```
8.0
<class 'float'>
```

Explicit Typecasting

Explicit typecasting is performed when the programmer explicitly converts one data type to another using built-in functions. This is also known as type conversion.

Common Typecasting Functions

- int(): Converts a value to an integer.
- float(): Converts a value to a float.
- str(): Converts a value to a string.
- list(): Converts a value to a list.
- tuple(): Converts a value to a tuple.
- set(): Converts a value to a set.
- dict(): Converts a value to a dictionary (from a list of key-value pairs).
- bool(): Converts a value to a boolean.

Example

- **int()**

```
# String to Integer
a = "10"
```

```
b = int(a)
print(b)  # Output: 10
print(type(b))  # Output: <class 'int'>

# Float to Integer
c = 10.5
d = int(c)
print(d)  # Output: 10
print(type(d))  # Output: <class 'int'>
```

```
10
<class 'int'>
10
<class 'int'>
```

- **float()**

```
# String to Float
e = "12.34"
f = float(e)
print(f)  # Output: 12.34
print(type(f))  # Output: <class 'float'>

# Integer to Float
g = 5
h = float(g)
print(h)  # Output: 5.0
print(type(h))  # Output: <class 'float'>
```

```
12.34
<class 'float'>
5.0
<class 'float'>
```

- **str()**

```
# Integer to String
i = 123
j = str(i)
print(j)  # Output: '123'
print(type(j))  # Output: <class 'str'>

# Float to String
k = 45.67
l = str(k)
print(l)  # Output: '45.67'
print(type(l))  # Output: <class 'str'>
```

```
123
<class 'str'>
45.67
<class 'str'>
```

- **list()**

```
# String to List
m = "hello"
n = list(m)
print(n)  # Output: ['h', 'e', 'l', 'l', 'o']
print(type(n))  # Output: <class 'list'>

# Tuple to List
o = (1, 2, 3)
p = list(o)
print(p)  # Output: [1, 2, 3]
print(type(p))  # Output: <class 'list'>
```

```
['h', 'e', 'l', 'l', 'o']
<class 'list'>
[1, 2, 3]
```

```
<class 'list'>
```

- **tuple()**

```
# List to Tuple
q = [1, 2, 3]
r = tuple(q)
print(r)  # Output: (1, 2, 3)
print(type(r))  # Output: <class 'tuple'>

# String to Tuple
s = "hello"
t = tuple(s)
print(t)  # Output: ('h', 'e', 'l', 'l', 'o')
print(type(t))  # Output: <class 'tuple'>
```

```
(1, 2, 3)
<class 'tuple'>
('h', 'e', 'l', 'l', 'o')
<class 'tuple'>
```

- **set()**

```
# List to Set
u = [1, 2, 2, 3]
v = set(u)
print(v)  # Output: {1, 2, 3}
print(type(v))  # Output: <class 'set'>

# String to Set
w = "hello"
x = set(w)
print(x)  # Output: {'h', 'e', 'l', 'o'}
print(type(x))  # Output: <class 'set'>
```

```
{1, 2, 3}
<class 'set'>
{'e', 'o', 'h', 'l'}
<class 'set'>
```

- **dict()**

```
# List of Tuples to Dictionary
y = [('a', 1), ('b', 2)]
z = dict(y)
print(z)  # Output: {'a': 1, 'b': 2}
print(type(z))  # Output: <class 'dict'>
```