Module 1

Python Basic Concepts and Programming

Introduction to Python

***Python is an object-oriented, high level language, interpreted, dynamic and multipurpose programming language.**

***Python is not intended to work on special area such as web programming. That is why it is known as multipurpose because it can be used with web, enterprise.**

***We don’t need to use data types to declare variable because it is dynamically typed so we can write a=10 to declare an integer value in a variable.**

***Python makes the development and debugging fast because there is no compilation step included in python development.**

History

Python was first introduced by **Guido Van Rossum** in **1991** at the **National Research Institute for Mathematics and Computer Science, Netherlands.**

Though the language was introduced in 1991, the development began in the 1980s. Previously van Rossum worked on the ABC language at CentrumWiskunde & Informatica (CWI) in the Netherlands.

The ABC language was capable of exception handling and interfacing with the Amoeba operating system. Inspired by the language, Van Rossum first tried out making his own version of it.

Python is influenced by programming languages like: ABC language, Modula-3, Python is used for software development at companies and organizations such as Google, Yahoo, CERN, Industrial Light and Magic, and NASA.

Why the Name Python?

Python developer, Rossum always wanted the name of his new language to be short, unique, and mysterious. Inspired by Monty Python’s Flying Circus, a BBC comedy series, he named it Python.

Features of Python

1. Easy to Code:

Python is a very developer-friendly language which means that anyone and everyone can learn to code it in a couple of hours or days. As compared to other object-oriented programming languages like Java, C, C++, and C#, Python is one of the easiest to learn.

2. Open Source and Free:

Python is an open-source programming language which means that anyone can create and contribute to its development. Python has an online forum where thousands of coders gather daily to improve this language further. Along with this Python is free to download and use in any operating system, be it Windows, Mac or Linux.

3. Support for GUI:

GUI or Graphical User Interface is one of the key aspects of any programming language because it has the ability to add flair to code and make the results more visual. Python has support for a wide array of GUIs which can easily be imported to the interpreter, thus making this one of the most favorite languages for developers.

4. Object-Oriented Approach:

One of the key aspects of Python is its object-oriented approach. This basically means that Python recognizes the concept of class and object encapsulation thus allowing programs to be efficient in the long run.

Features of Python

5. Highly Portable:

Suppose you are running Python on Windows and you need to shift the same to either a Mac or a Linux system, then you can easily achieve the same in Python without having to worry about changing the code.This is not possible in other programming languages, thus making Python one of the most portable languages available in the industry

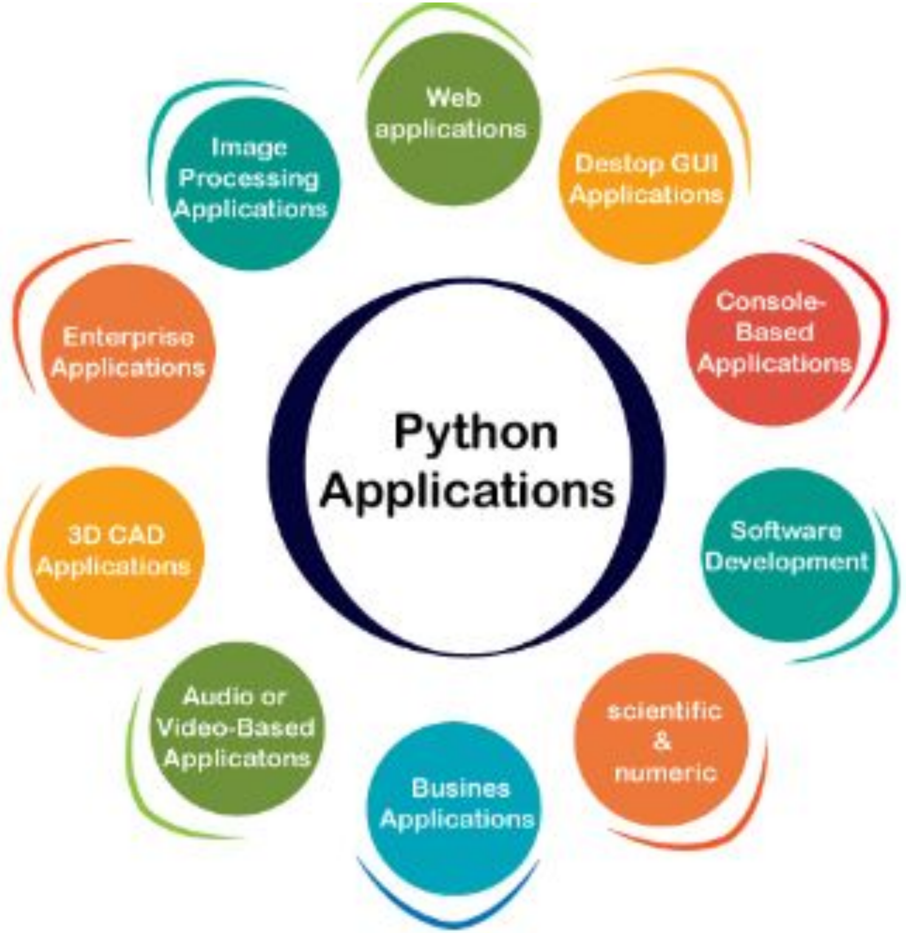
6. Highly Dynamic:

Python is one of the most dynamic languages available in the industry today. What this basically means is that the type of a variable is decided at the run time and not in advance. Due to the presence of this feature, we do not need to specify the type of the variable during coding, thus saving time and increasing efficiency.

7. Large Standard Library:

Out of the box, Python comes inbuilt with a large number of libraries that can be imported at any instance and be used in a specific program. The presence of libraries also makes sure that you don’t need to write all the code yourself and can import the same from those that already exist in the libraries.

Applications of Python



Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Here, we are specifying application areas where Python can be applied.

Applications of Python

1) Web Applications

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing

One of Python web-framework named Django is used on **Instagram**. Python provides many useful frameworks, and these are given below:

- Django and Pyramid framework(Use for heavy applications)

2) Desktop GUI Applications

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a (Tkinter)**Tk GUI library** to develop a user interface.

3)3D CAD Applications

The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using the following functionalities.

- Fandango (Popular)
- AnyCAD

4)Image Processing Application

Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing are given below.

- OpenCV
- Pillow

5)Enterprise Applications

Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

6)Audio or Video-based Applications

Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are **TimPlayer, cplay,** etc. The few multimedia libraries are given below.

- Gstreamer

7) Scientific and Numeric

Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.

Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy,Matplotlib

Introducing the Python Interpreter

An interpreter is a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

When the Python package is installed on your machine, it generates a number of components—minimally, an interpreter and a support library. Depending on how you use it, the Python interpreter may take the form of an executable program.

Python code you write must always be run by this interpreter. And to enable that, you must install a Python interpreter on your computer.

Program Execution

The execution of the Python program involves 2 Steps:

- Compilation
- Interpreter

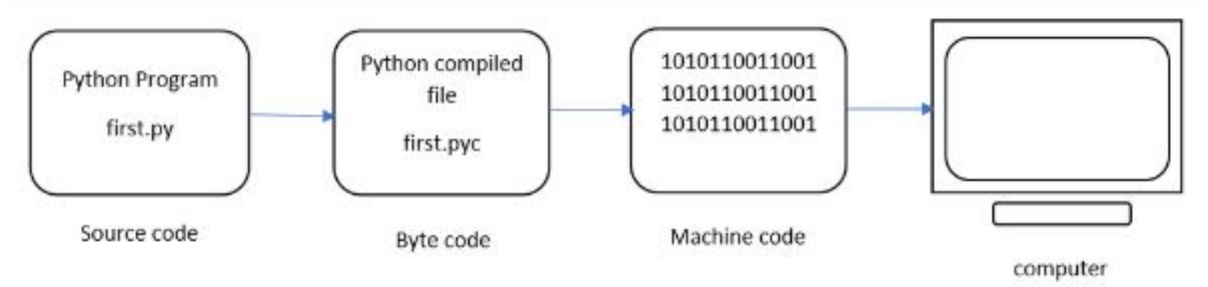
when you execute a program Python first compiles your source code into a format known as byte code.Compilation is simply a translation step, and byte code is a lower-level,platform-independent representation of your source code.It can run on any operating system and hardware. The byte code instructions are created in the **.pyc** file.

Python instead saves its.pyc byte code files in a subdirectory named __pycache__ located in the directory where your source files reside, and in files whose names identify the Python version that created them (e.g., script.cpython-33.pyc)

Interpreter

The next step involves converting the byte code (.pyc file) into machine code. This step is necessary as the computer can understand only machine code (binary code). Python Virtual Machine (PVM) first understands the operating system and processor in the computer and then converts it into machine code. Further, these machine code instructions are executed by processor and the results are displayed.

However, the interpreter inside the PVM translates the program line by line thereby consuming a lot of time. To overcome this, a compiler known as Just In Time (JIT) is added to PVM. JIT compiler improves the execution speed of the Python program. This compiler is not used in all Python environments like CPython which is standard Python software.



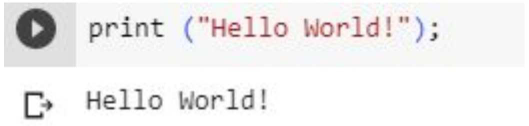
PVM

Once your program has been compiled to byte code it is shipped off for execution to something generally known as the Python Virtual Machine

PVM is just a big code loop that iterates through your byte code instructions, one by one, to carry out their operations. The PVM is the runtime engine of Python; it’s always present as part of the Python system, and it’s the component that truly runs your scripts. Technically, it’s just the last step of what is called the “Python interpreter.”

Statements

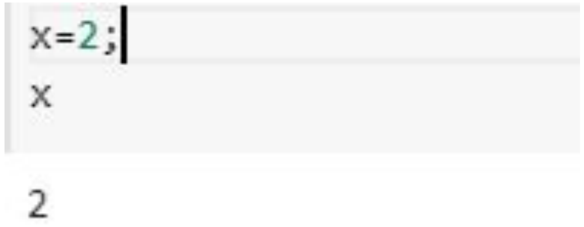
- Any instruction written in the source code and executed by the Python interpreter is called a statement.
- We have 2 kinds of statements 1)print 2)assignment
- 1)print statement:Python executes it and displays the result.the result of print statement is a value.



```
print ("Hello World!");
```

```
Hello World!
```

- 2)Assignment statement:Assignment statement dont produce result

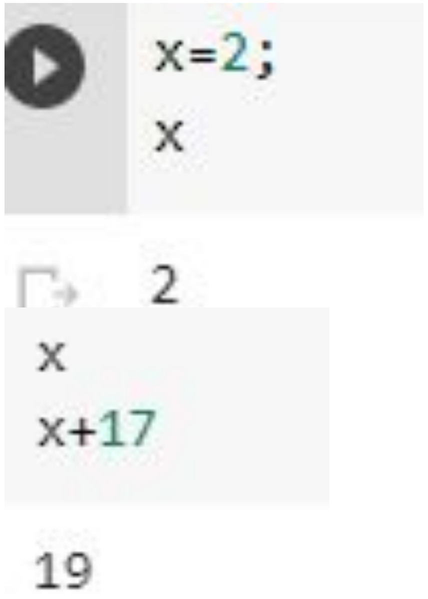


```
x=2;
x
```

```
2
```

Expressions

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable , so the following are all legal expressions.Operand is value on which operator is applied.These operators use constants and variables to form an expression



```
x=2;
x
```

```
2
```

```
x
x+17
```

```
19
```

Types of Expressions

Based on the position of operators in an expression

1)Constant expression:One that involves only constants eg: 8+9-2

2)Integral expression:One that produces integer result after evaluating the expression.eg:

a=10

b=5

c=a*b

Types of Expressions

3)Floating point expression: One that produces floating point results.Eg:a*b/2

4)relational expression:One that returns either True or False eg: c=a>b

5)Logical expressions:One that combines two or more relational expressions and returns a value as True or False.Eg:a>b && y!=0

6)Bitwise expressions:One that manipulates data at bit level.Eg: x=y&z

7)Assignment expressions:One that assigns a value to a variable.eg: c=a + b or c=10

Types of Expressions

Based on the position of operators in an expression.

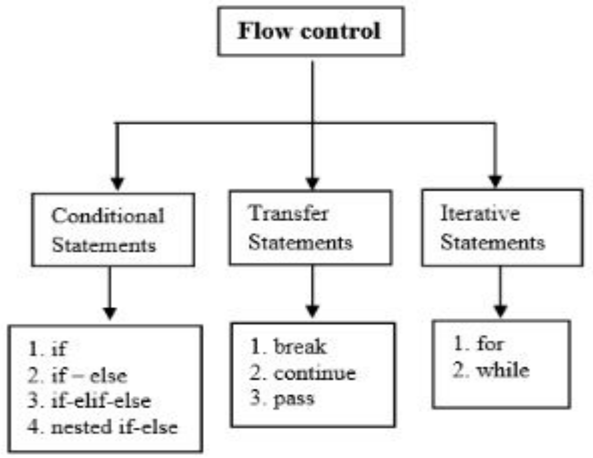
1)Infix expression:In which operator is placed in between the operands eg:a=b-c

2)Prefix expression: The operator is placed before the operands.eg:a=+bc

3)Postfix expression: The operator is placed after the operands.eg:a=bc+

Control Flow Statements:

flow control is the order in which statements or blocks of code are executed at runtime based on a condition.



The flow control statements are divided into **three categories**

1. Conditional statements
2. Iterative statements.
3. Transfer statements

Conditional statements

In Python, condition statements act depending on whether a given condition is true or false. You can execute different blocks of codes depending on the outcome of a condition. Condition statements always evaluate to either True or False.There are three types of conditional statements

1. if statement
2. if-else
3. if-elif-else
4. nested if-else

Control Flow Statements

Iterative statements

In Python, iterative statements allow us to execute a block of code repeatedly as long as the condition is True. We also call it a loop statements.Python provides us the following two loop statement to perform some actions repeatedly

1. for loop
2. while loop

Transfer statements

In Python, transfer statements are used to alter the program’s way of execution in a certain manner. For this purpose, we use three types of transfer statements.

1. break statement
2. continue statement

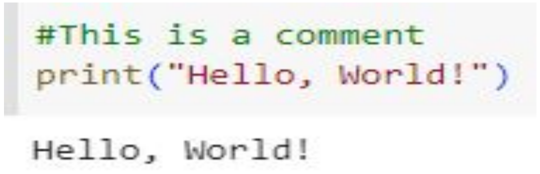
Parts of Python Programming Language

Comments:Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Creating a Comment

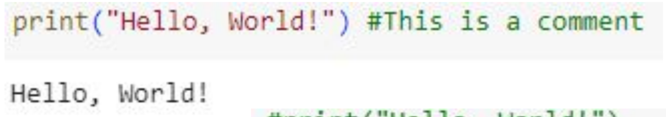
Comments starts with a #, and Python will ignore them:



```
#This is a comment
print("Hello, World!")
```

```
Hello, World!
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

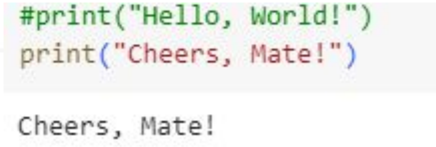


```
print("Hello, World!") #This is a comment
```

```
Hello, World!
```

A comment does not have to be text that explains the code, it can also be used to

 prevent Python from executing code:

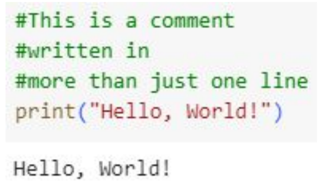


```
#print("Hello, World!")
print("Cheers, Mate!")
```

```
Cheers, Mate!
```

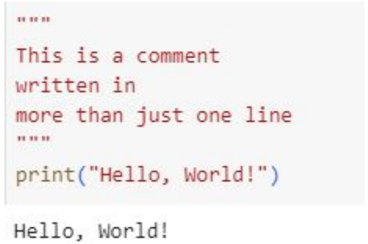
Multiline Comments

Python does not really have a syntax for multiline comments.To add a multiline comment you could insert a # for each line:Or, not quite as intended, you can use a multiline string.Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:



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

```
Hello, World!
```



```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

```
Hello, World!
```

Variables

Variables are containers for storing data values.

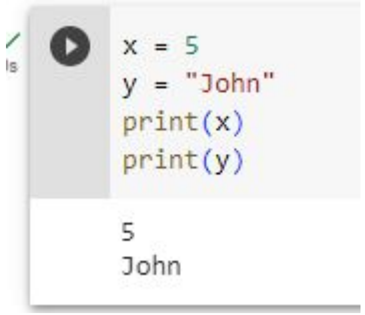
Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

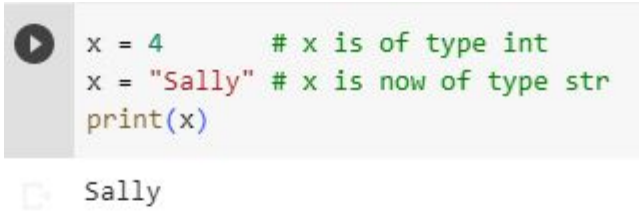
Variables do not need to be declared with any particular *type*, and

can even change type after they have been set.



```python
x = 5
y = "John"
print(x)
print(y)
```

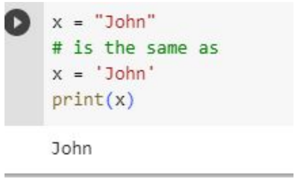
```
5
John
```



```python
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

```
Sally
```

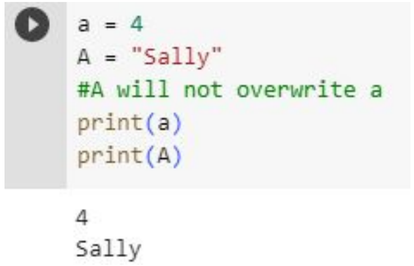
String variables can be declared either by using single or double quotes:



```python
x = "John"
# is the same as
x = 'John'
print(x)
```

```
John
```

Case-Sensitive: Variable names are case-sensitive.

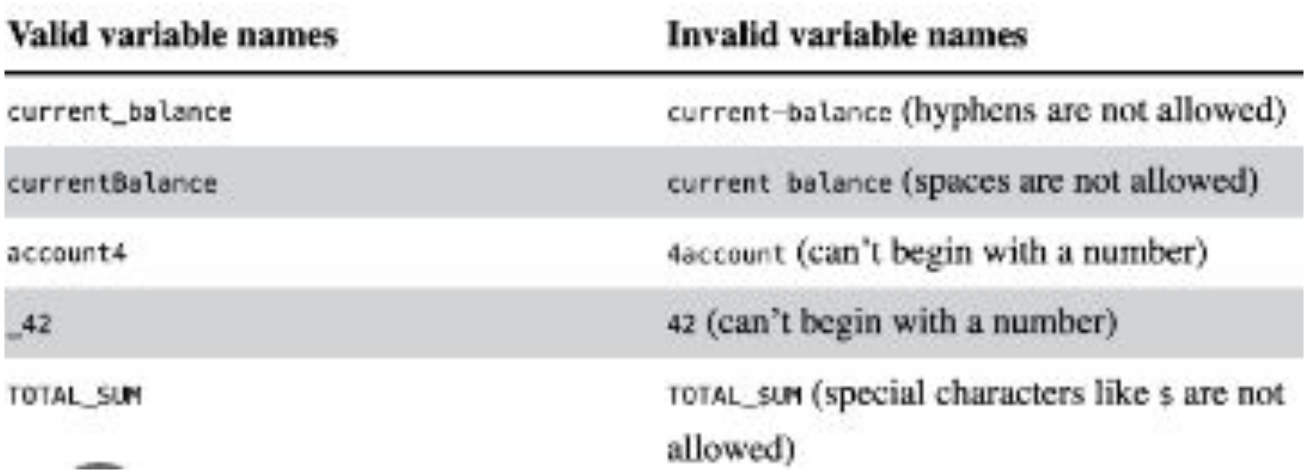


```python
a = 4
A = "Sally"
#A will not overwrite a
print(a)
print(A)
```

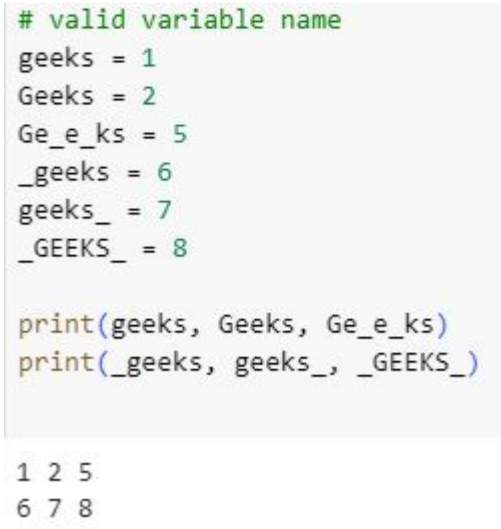
```
4
Sally
```

Rules for Python variables

- A Python variable name must start with a letter or the underscore character.
- A Python variable name cannot start with a number.
- A Python variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable in Python names are case-sensitive (name, Name, and NAME are three different variables).
- The reserved words(keywords) in Python cannot be used to name the variable in Python.



Valid variable names	Invalid variable names
current_balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
account4	4account (can't begin with a number)
_42	42 (can't begin with a number)
TOTAL_SUM	TOTAL_$UM (special characters like $ are not allowed)



```
# valid variable name
geeks = 1
Geeks = 2
Ge_e_ks = 5
_geeks = 6
geeks_ = 7
_GEEKS_ = 8

print(geeks, Geeks, Ge_e_ks)
print(_geeks, geeks_, _GEEKS_)
```

```
1 2 5
6 7 8
```

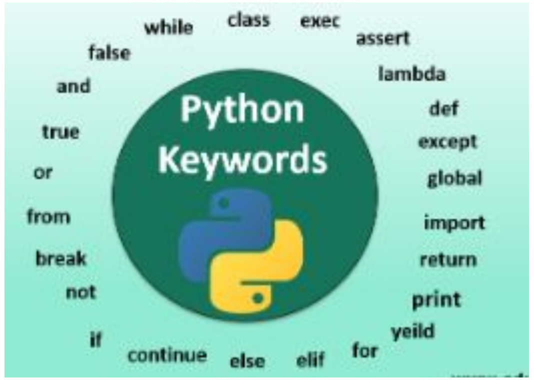
Keywords in Python

- **Python Keywords** are some predefined and reserved words in python that have special meanings
- Python keywords cannot be used as the name of variables, functions, and classes.
- Python programming language, there are a set of predefined words, called Keywords which along with Identifiers will form meaningful sentences when used together.

Rules for Keywords in Python

- Python keywords cannot be used as identifiers.
- All the keywords in python should be in lowercase except True and False.

List of Python Keywords



Keywords in Python

The following code allows you to view the complete list of Python’s keywords.

This code imports the “keyword” module in Python and then prints a list of all the keywords in Python using the “kwlist” attribute of the “keyword” module. The “kwlist” attribute is a list of strings, where each string represents a keyword in Python. By printing this list, we can see all the keywords that are reserved in Python and cannot be used as identifiers.

```python
# code
import keyword

print(keyword.kwlist)
```

Identifiers in Python

- **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.
- It is a good programming practice to give meaningful names to identifiers to make the code understandable.

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 (_).

Examples of Python Identifiers

Valid identifiers:*var1 , _var1, _1_var*

Invalid Identifiers:*!var1 , 1var ,1_var*

Operators

Operators are the construct used to manipulate values of operands

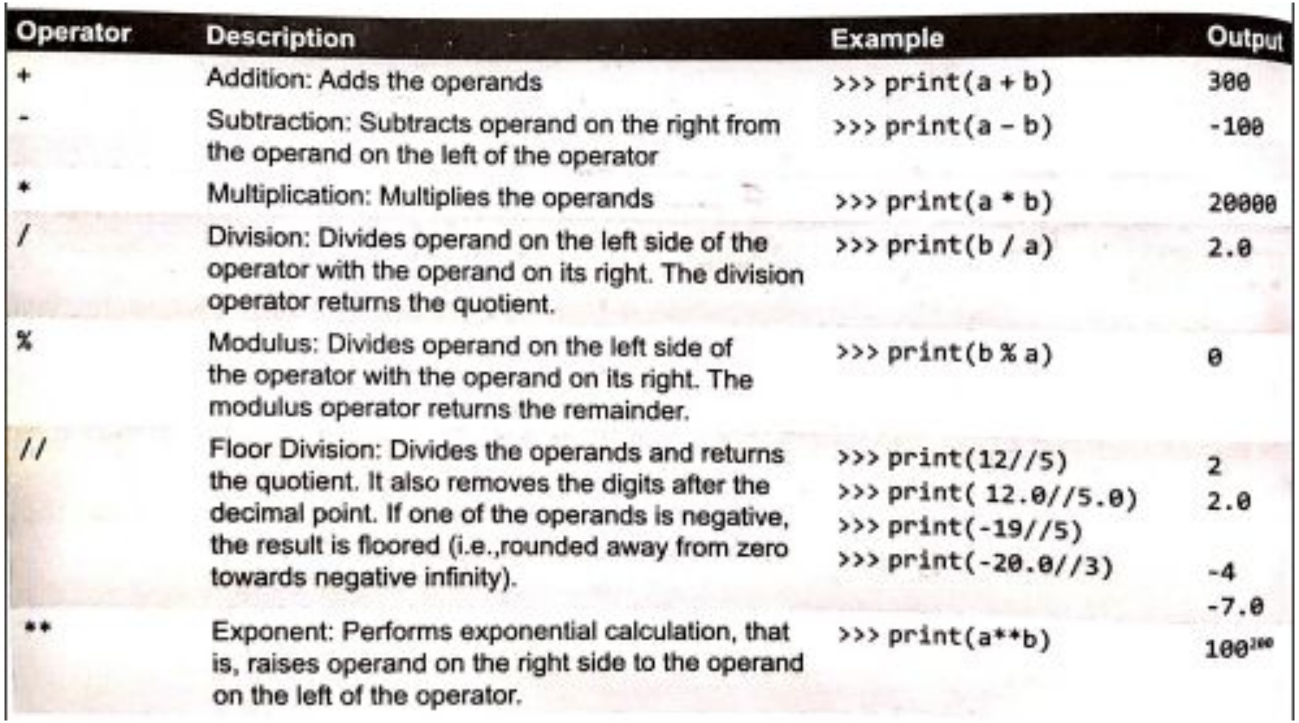
Some basic operators include *,-,+,/

Python supports different types of operators

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

Arithmetic Operators

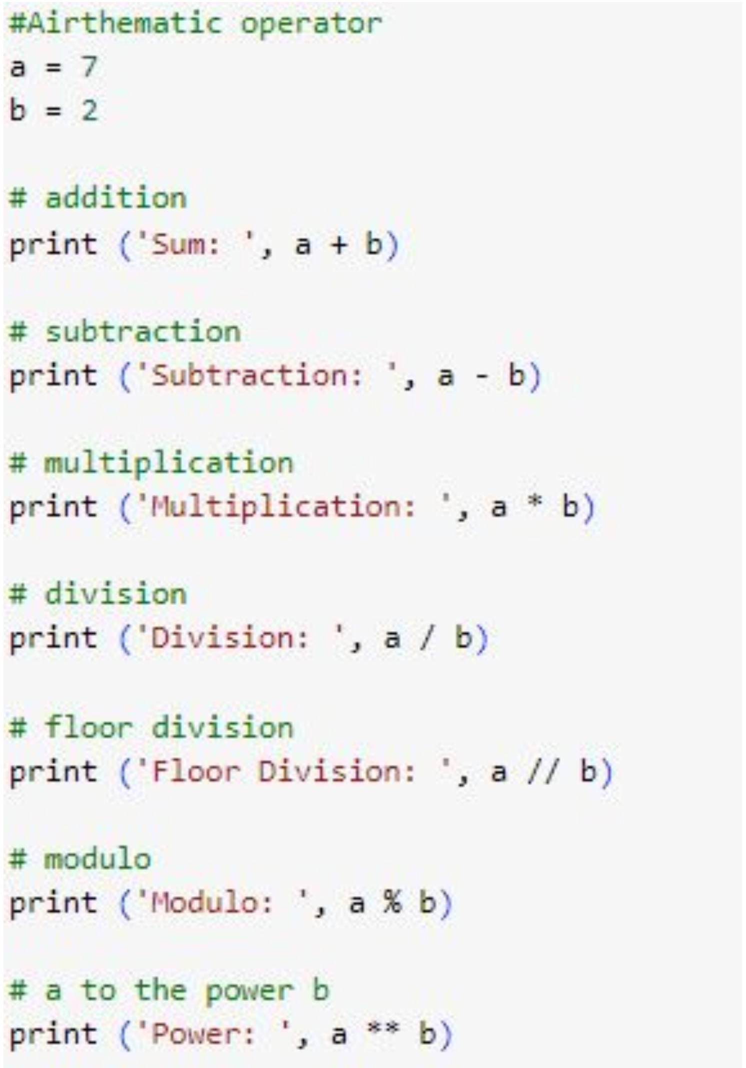
Arithmetic operators are used with numeric values to perform common mathematical operations:



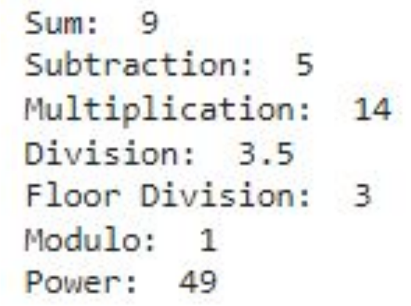
Operator	Description	Example	Output
+	Addition: Adds the operands	`>>> print(a + b)`	300
-	Subtraction: Subtracts operand on the right from the operand on the left of the operator	`>>> print(a - b)`	-100
*	Multiplication: Multiplies the operands	`>>> print(a * b)`	20000
/	Division: Divides operand on the left side of the operator with the operand on its right. The division operator returns the quotient.	`>>> print(b / a)`	2.0
%	Modulus: Divides operand on the left side of the operator with the operand on its right. The modulus operator returns the remainder.	`>>> print(b % a)`	0

//	Floor Division: Divides the operands and returns the quotient. It also removes the digits after the decimal point. If one of the operands is negative, the result is floored (i.e.,rounded away from zero towards negative infinity).	`>>> print(12//5)`
		`>>> print( 12.0//5.0)`
		`>>> print(-19//5)`
		`>>> print(-20.0//3)`
			2
			2.0
			-4
			-7.0

**	Exponent: Performs exponential calculation, that is, raises operand on the right side to the operand on the left of the operator.	`>>> print(a**b)`	100^{200}



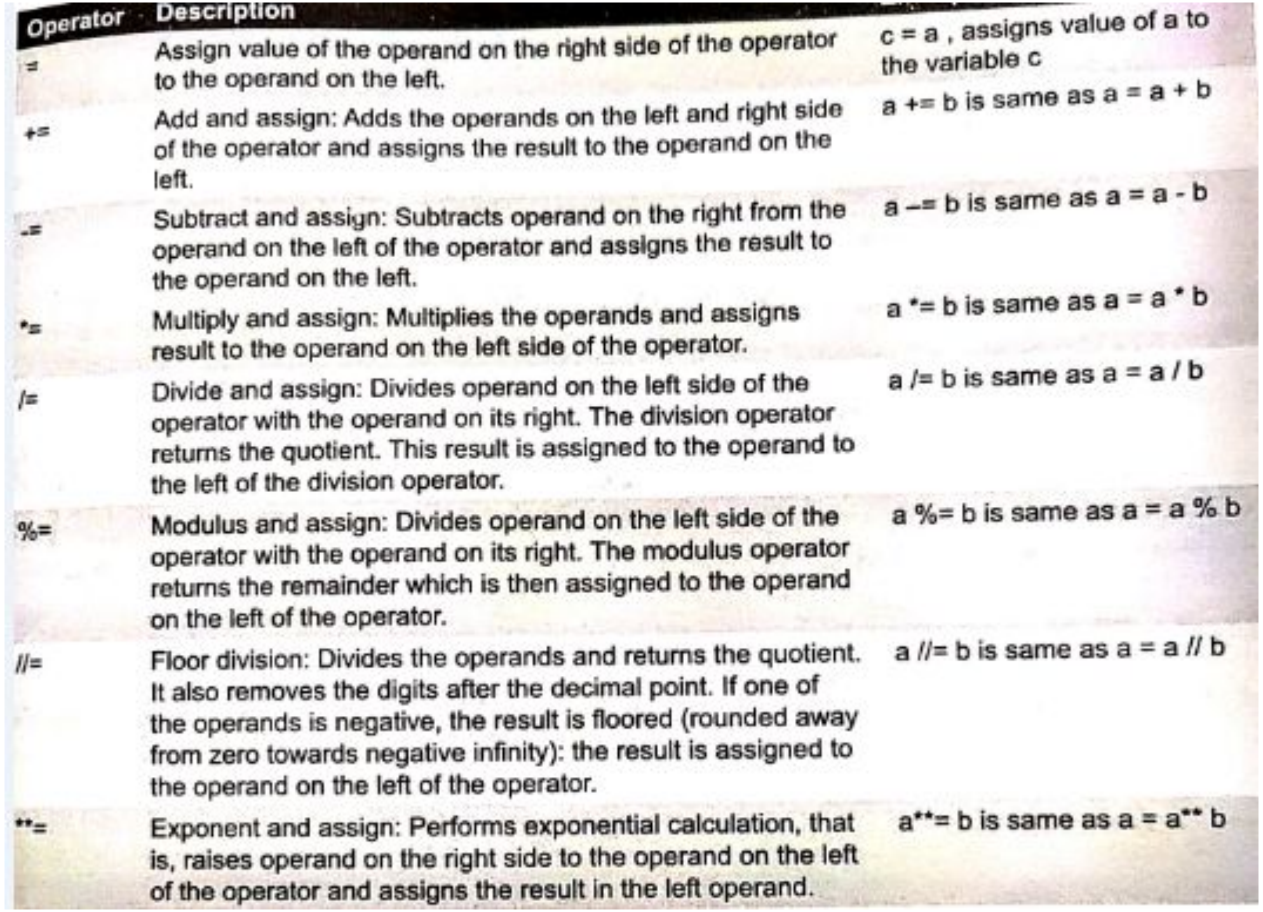
```python
#Airthematic operator
a = 7
b = 2

# addition
print ('Sum: ', a + b)

# subtraction
print ('Subtraction: ', a - b)

# multiplication
print ('Multiplication: ', a * b)

# division
print ('Division: ', a / b)

# floor division
print ('Floor Division: ', a // b)

# modulo
print ('Modulo: ', a % b)

# a to the power b
print ('Power: ', a ** b)
```



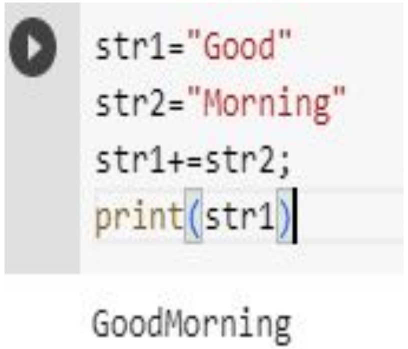
```
Sum:  9
Subtraction:  5
Multiplication:  14
Division:  3.5
Floor Division:  3
Modulo:  1
Power:  49
```

Assignment Operators:

Assignment operators are used to assign values to variables:

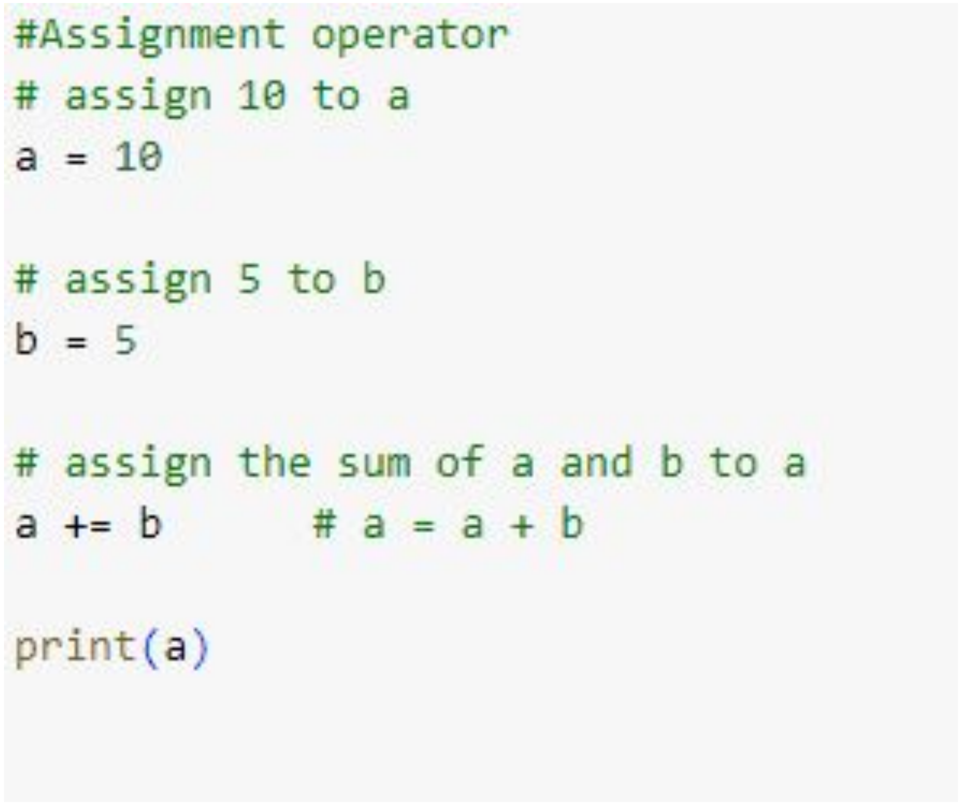


Operator	Description	
=	Assign value of the operand on the right side of the operator to the operand on the left.	c = a , assigns value of a to the variable c
+=	Add and assign: Adds the operands on the left and right side of the operator and assigns the result to the operand on the left.	a += b is same as a = a + b
-=	Subtract and assign: Subtracts operand on the right from the operand on the left of the operator and assigns the result to the operand on the left.	a –= b is same as a = a - b
*=	Multiply and assign: Multiplies the operands and assigns result to the operand on the left side of the operator.	a *= b is same as a = a * b
/=	Divide and assign: Divides operand on the left side of the operator with the operand on its right. The division operator returns the quotient. This result is assigned to the operand to the left of the division operator.	a /= b is same as a = a / b
%=	Modulus and assign: Divides operand on the left side of the operator with the operand on its right. The modulus operator returns the remainder which is then assigned to the operand on the left of the operator.	a %= b is same as a = a % b
//=	Floor division: Divides the operands and returns the quotient. It also removes the digits after the decimal point. If one of the operands is negative, the result is floored (rounded away from zero towards negative infinity): the result is assigned to the operand on the left of the operator.	a //= b is same as a = a // b
**=	Exponent and assign: Performs exponential calculation, that is, raises operand on the right side to the operand on the left of the operator and assigns the result in the left operand.	a**= b is same as a = a** b



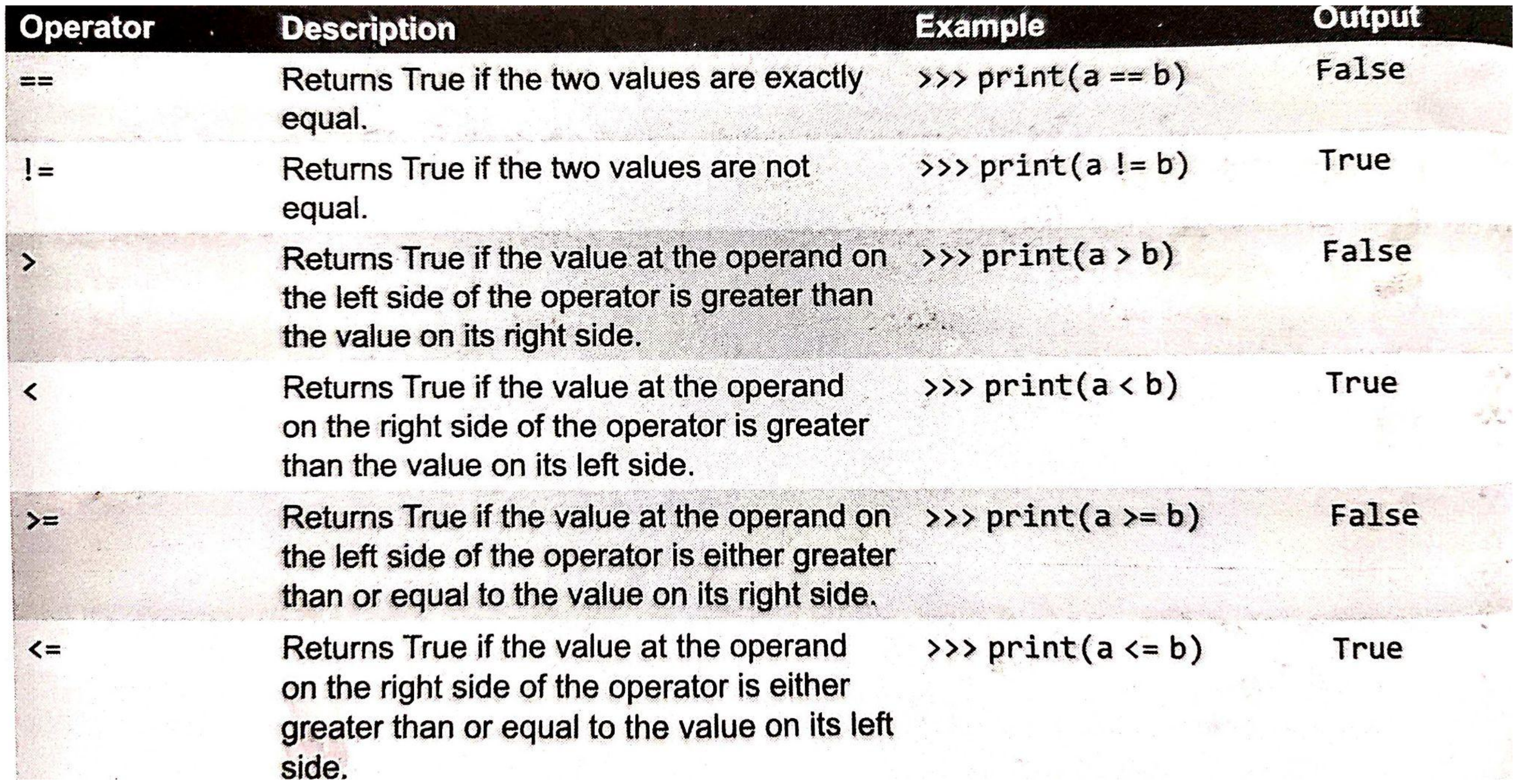
```
str1="Good"
str2="Morning"
str1+=str2;
print(str1)
```

```
GoodMorning
```

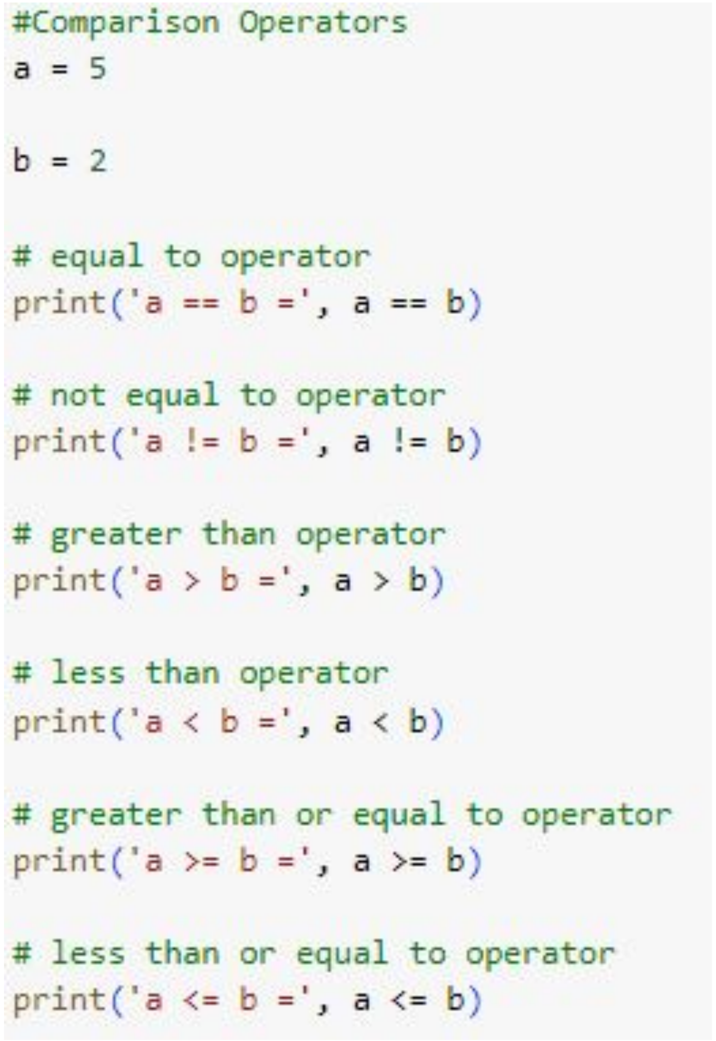


```python
#Assignment operator
# assign 10 to a
a = 10

# assign 5 to b
b = 5

# assign the sum of a and b to a
a += b      # a = a + b

print(a)
```

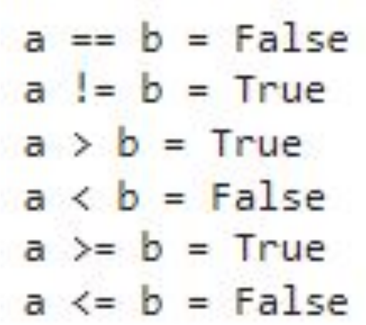
Comparison Operators: Comparison operators are used to compare two values



Operator	Description	Example	Output
==	Returns True if the two values are exactly equal.	`>>> print(a == b)`	False
!=	Returns True if the two values are not equal.	`>>> print(a != b)`	True
>	Returns True if the value at the operand on the left side of the operator is greater than the value on its right side.	`>>> print(a > b)`	False
<	Returns True if the value at the operand on the right side of the operator is greater than the value on its left side.	`>>> print(a < b)`	True
>=	Returns True if the value at the operand on the left side of the operator is either greater than or equal to the value on its right side.	`>>> print(a >= b)`	False
<=	Returns True if the value at the operand on the right side of the operator is either greater than or equal to the value on its left side.	`>>> print(a <= b)`	True



```python
#Comparison Operators
a = 5

b = 2

# equal to operator
print('a == b =', a == b)

# not equal to operator
print('a != b =', a != b)

# greater than operator
print('a > b =', a > b)

# less than operator
print('a < b =', a < b)

# greater than or equal to operator
print('a >= b =', a >= b)

# less than or equal to operator
print('a <= b =', a <= b)
```



```
a == b = False
a != b = True
a > b = True
a < b = False
a >= b = True
a <= b = False
```

Logical Operators: Logical operators are used to combine conditional statements

1)Logical AND
 Logical AND is used to simultaneously evaluate two conditions or expressions with relational operators.
The whole expression is true only if both the expressions are true

2)Logical OR
Logical OR is used to simultaneously evaluate two conditions or expressions with relational operators.
The whole expression is true only if one or both the expressions of the logical operator is true

3)Logical NOT
The logical NOT operator takes a single expression and negates the value of the expression

```python
# Examples of Logical Operator

a = True
b = False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

Output

```
False
True
False
```

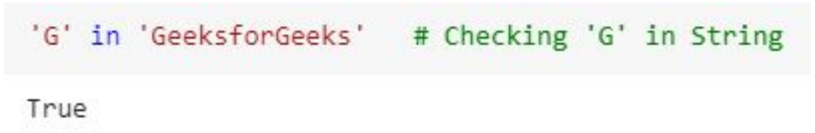
Membership Operators

Python offers two membership operators to check or validate the membership of a value.

It tests for membership in a sequence, such as strings, lists, or tuples.

in operator: The ‘in’ operator is used to check if a character/ substring/ element exists in a sequence or not. Evaluate to True if it finds the specified element in a sequence otherwise False.

Example

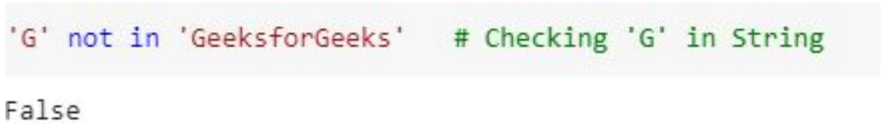


```
'G' in 'GeeksforGeeks'   # Checking 'G' in String
```

```
True
```

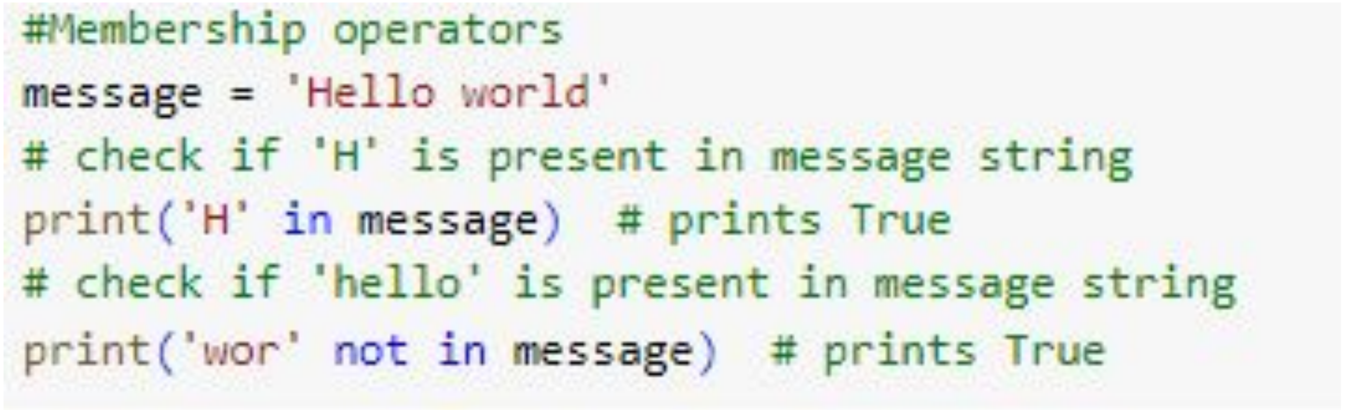
‘not in’ operator- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Example

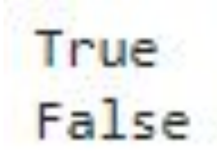


```
'G' not in 'GeeksforGeeks'   # Checking 'G' in String
```

```
False
```



```
#Membership operators
message = 'Hello world'
# check if 'H' is present in message string
print('H' in message)  # prints True
# check if 'hello' is present in message string
print('wor' not in message)  # prints True
```



```
True
False
```

Identity Operators

Identity operators are used to compare the objects if both the objects are actually of the same data type and share the same memory location.

There are different identity operators such as

‘is’ operator – Evaluates to True if the variables on either side of the operator point to the same object and false otherwise.

# Python program to illustrate the use of 'is' identity operator

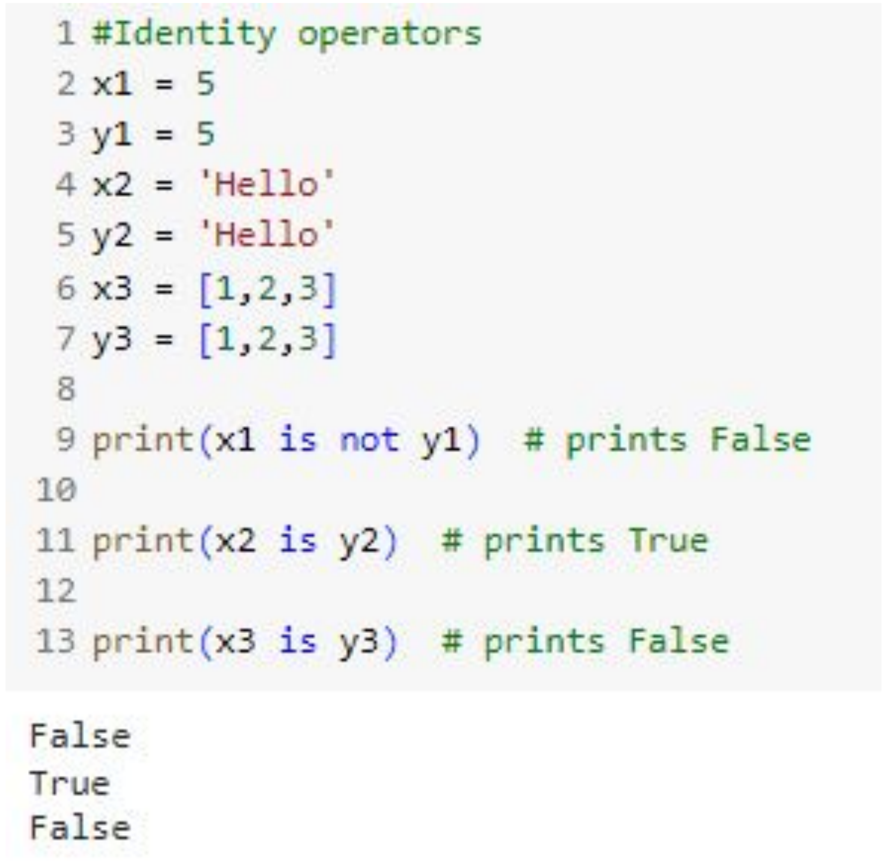
Example

```
x = 5

y = 6.0

print(x is y)
```

Identity Operators

‘is not’ operator: Evaluates True if both variables are not the same object.

```
x = ["Geeks", "for", "Geeks"]
y = ["Geeks", "for", "Geeks"]
z = x

# Returns False because z is the same object as x
print(x is not z)

# Returns True because x is not the same object as y,
# even if they have the same content
print(x is not y)

# To demonstrate the difference between "is not" and "!=":
# This comparison returns False because x is equal to y
print(x != y)
```



```python
#Identity operators
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

print(x1 is not y1)  # prints False

print(x2 is y2)  # prints True

print(x3 is y3)  # prints False
```

```
False
True
False
```

Bitwise Operators:

Python Bitwise operators act on bits and perform bit-by-bit operations.

In Python, bitwise operators are used to perform bitwise calculations on integers.

The integers are first converted into binary and then operations are performed on each bit or corresponding pair of bits, hence the name bitwise operators.

The result is then returned in decimal format.

1)Bitwise AND(&):Result bit 1,if all of the operand bit is 1; otherwise results bit 0.

2)Bitwise OR(|):Result bit 1,if any of the operand bit is 1; otherwise results bit 0.

3)Bitwise NOT(^):inverts individual bits

4)Bitwise XOR:Results bit 1,if any of the operand bit is 1 but not both, otherwise results bit 0

5)Bitwise right shift:The left operand's value is moved toward right by the number of bits specified by the right operand.

6)Bitwise left shift:The left operand's value is moved toward left by the number of bits

specified by the right operand.

Bitwise AND Operator

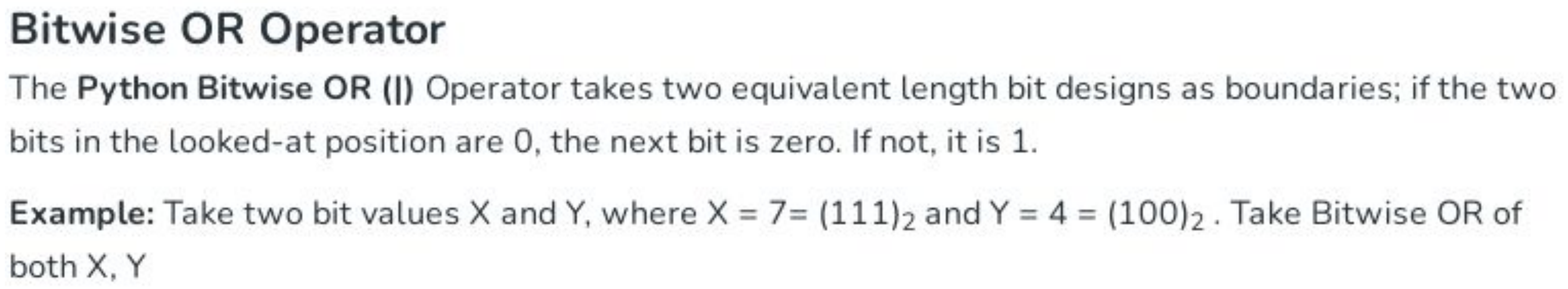
The **Python Bitwise AND (&)** operator takes two equal-length bit patterns as parameters. The two-bit integers are compared. If the bits in the compared positions of the bit patterns are 1, then the resulting bit is 1. If not, it is 0.

Example: Take two bit values X and Y, where X = 7= $(111)_2$ and Y = 4 = $(100)_2$. Take Bitwise and of both X & y

Note: Here, $(111)_2$ represent binary number.

$$\begin{array}{r} 1\,1\,1_2 \\ \&\,1\,0\,0_2 \\ \hline 1\,0\,0_2 \;=\; 4 \end{array}$$

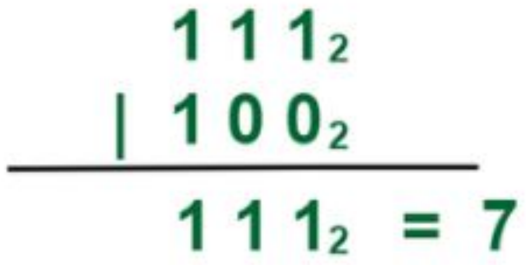
```
a = 10
b = 4

# Print bitwise AND operation
print("a & b =", a & b)
```



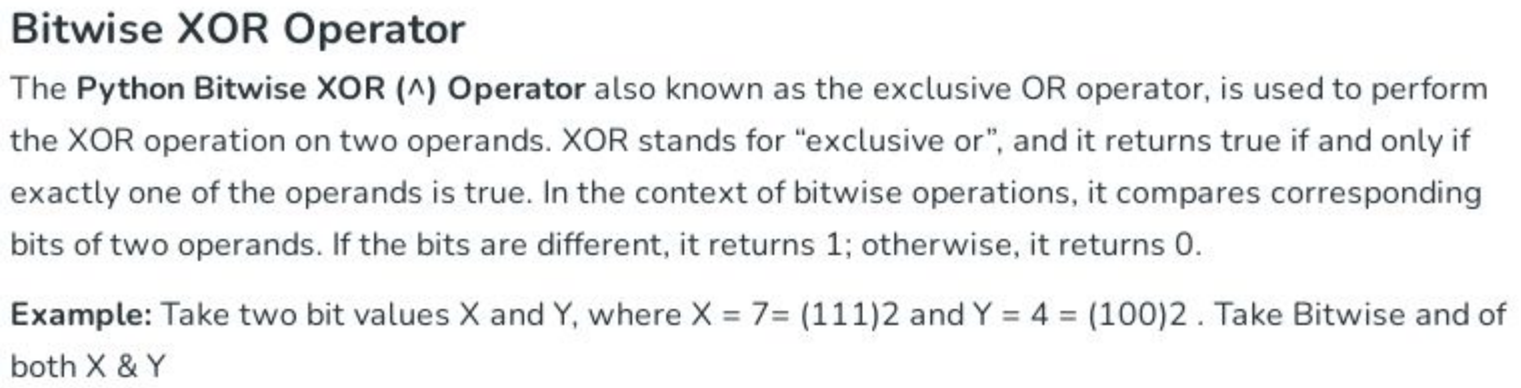
Bitwise OR Operator

The **Python Bitwise OR (|)** Operator takes two equivalent length bit designs as boundaries; if the two bits in the looked-at position are 0, the next bit is zero. If not, it is 1.

Example: Take two bit values X and Y, where X = 7= $(111)_2$ and Y = 4 = $(100)_2$. Take Bitwise OR of both X, Y



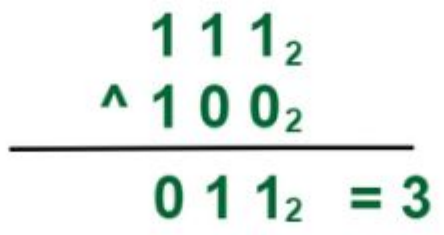
```
a = 10
b = 4

# Print bitwise OR operation
print("a | b =", a | b)
```



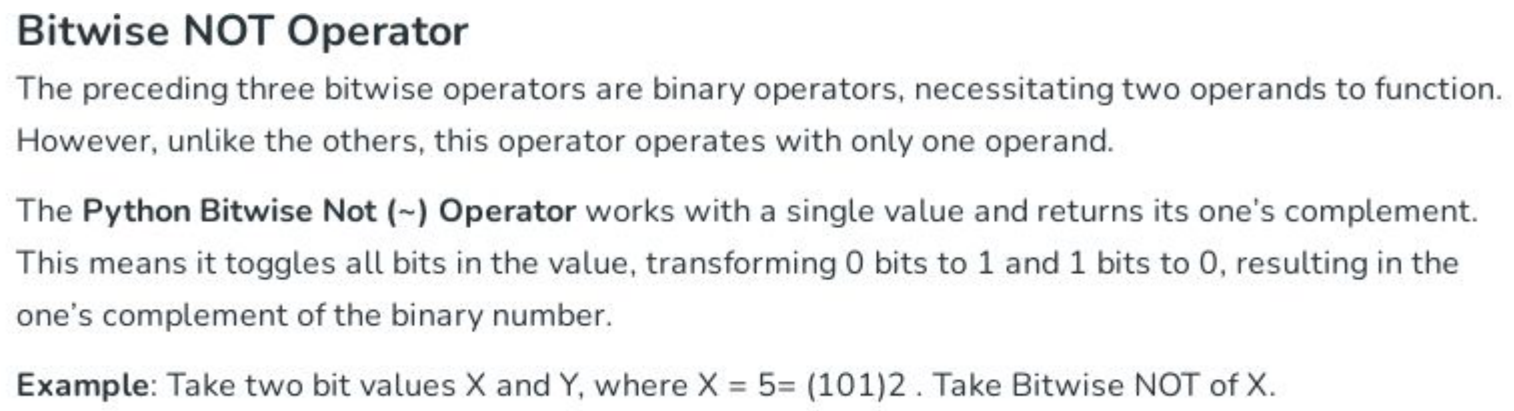
Bitwise XOR Operator

The **Python Bitwise XOR (^) Operator** also known as the exclusive OR operator, is used to perform the XOR operation on two operands. XOR stands for “exclusive or”, and it returns true if and only if exactly one of the operands is true. In the context of bitwise operations, it compares corresponding bits of two operands. If the bits are different, it returns 1; otherwise, it returns 0.

Example: Take two bit values X and Y, where X = 7= (111)2 and Y = 4 = (100)2 . Take Bitwise and of both X & Y



```
a = 10
b = 4

# print bitwise XOR operation
print("a ^ b =", a ^ b)
```

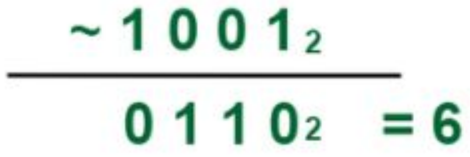


Bitwise NOT Operator

The preceding three bitwise operators are binary operators, necessitating two operands to function. However, unlike the others, this operator operates with only one operand.

The **Python Bitwise Not (~) Operator** works with a single value and returns its one's complement. This means it toggles all bits in the value, transforming 0 bits to 1 and 1 bits to 0, resulting in the one's complement of the binary number.

Example: Take two bit values X and Y, where X = 5= (101)2 . Take Bitwise NOT of X.

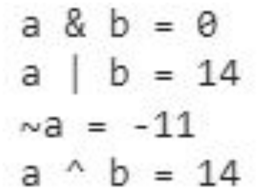


```
a = 10
b = 4

# Print bitwise NOT operation
print("~a =", ~a)
```

output:
```
~a = -11
```

Examples of Bitwise Operator

```python
# Python program to show
# bitwise operators

a = 10
b = 4

# Print bitwise AND operation
print("a & b =", a & b)

# Print bitwise OR operation
print("a | b =", a | b)

# Print bitwise NOT operation
print("~a =", ~a)

# print bitwise XOR operation
print("a ^ b =", a ^ b)
```

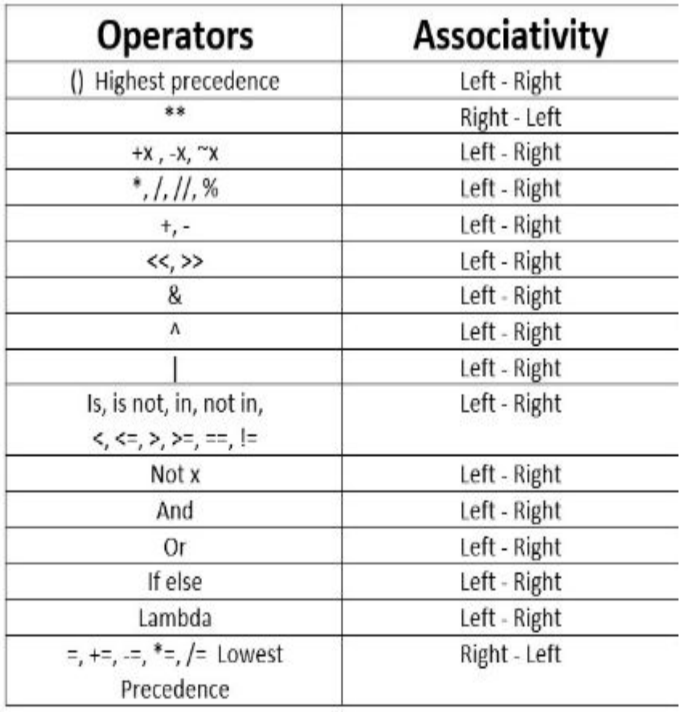


```
a & b = 0
a | b = 14
~a = -11
a ^ b = 14
```

Precedence and Associativity

Operator Precedence: This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

This table lists all operators from the highest precedence to the lowest precedence.



Operators	Associativity
() Highest precedence	Left - Right
**	Right - Left
+x , -x, ~x	Left - Right
*, /, //, %	Left - Right
+, -	Left - Right
<<, >>	Left - Right
&	Left - Right
^	Left - Right
|	Left - Right
Is, is not, in, not in, <, <=, >, >=, ==, !=	Left - Right
Not x	Left - Right
And	Left - Right
Or	Left - Right
If else	Left - Right
Lambda	Left - Right
=, +=, -=, *=, /= Lowest Precedence	Right - Left

Example

Example1:

```
expr = 10 + 20 * 30

print(expr)
```

Example2:

```
name = "Alex"
age = 0

if (name == "Alex" or name == "John") and age >= 2:
    print("Hello! Welcome.")
else:
    print("Good Bye!!")
```

Example3:

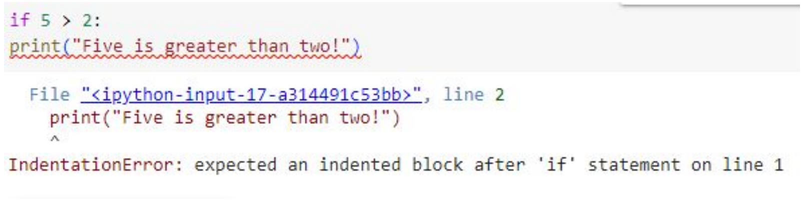
```
  100 + 200 / 10 - 3 * 10
```

Python Indentation

Indentation refers to the spaces at the beginning of a code line.Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.Python uses indentation to indicate a block of code.

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

Python will give you an error if you skip the indentation:



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

```
  File "<ipython-input-17-a314491c53bb>", line 2
    print("Five is greater than two!")
    ^
IndentationError: expected an indented block after 'if' statement on line 1
```

The number of spaces is up to you as a programmer, but it has to be at least one.

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

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

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

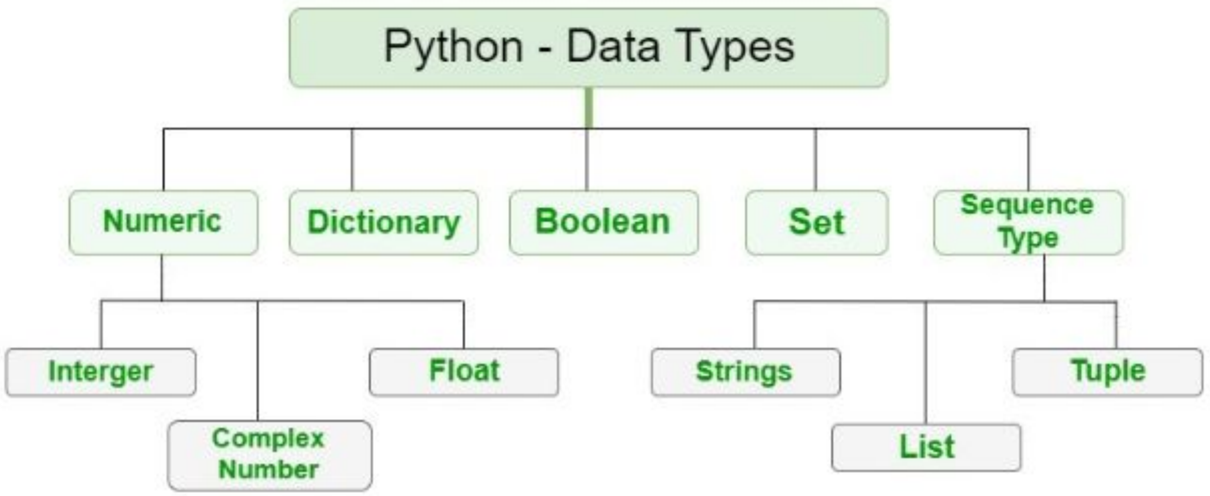
What is Python type() Function?

To define the values of various data types and check their data types we **use the type() function.** Consider the following examples.

```
# Python program to demonstrate numeric value

 a = 5
print("Type of a: ", type(a))
 b = 5.0
print("\nType of b: ", type(b))
 c = 2 + 4j
print("\nType of c: ", type(c))
```

Datatypes in Python

Data types are the classification or categorization of data items.
It represents the kind of value that tells what operations can be performed on a particular data.
Since everything is an object in Python programming, data types are actually classes and variables are instances (object) of these classes.
The following are the standard or built-in data types in Python:

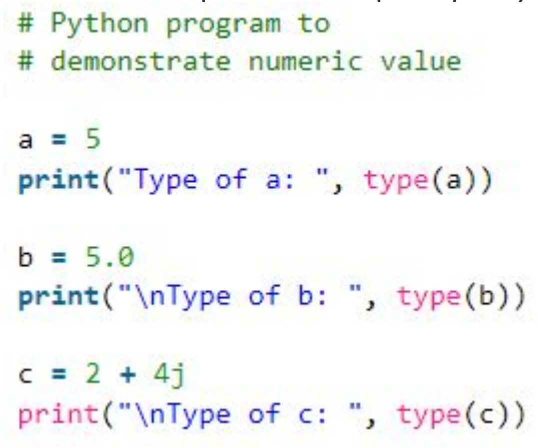


Numeric Data Type in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python int, Python float, and Python complex classes in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.
- **Float** – This value is represented by the float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** – Complex number is represented by a complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

Note – type() function is used to determine the type of data type.

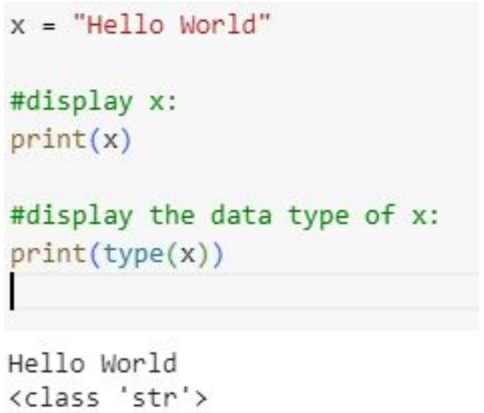


```
# Python program to
# demonstrate numeric value

a = 5
print("Type of a: ", type(a))

b = 5.0
print("\nType of b: ", type(b))

c = 2 + 4j
print("\nType of c: ", type(c))
```

Sequence Data Type in Python

The sequence Data Type in Python is the ordered collection of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence types in Python –

- Python String
- Python List
- Python Tuple



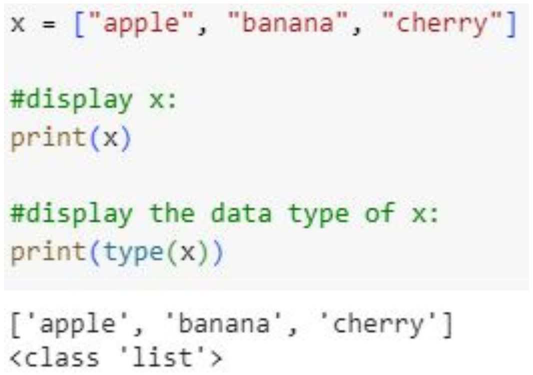
```
x = "Hello World"

#display x:
print(x)

#display the data type of x:
print(type(x))
```

```
Hello World
<class 'str'>
```

String Data Type: Strings in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

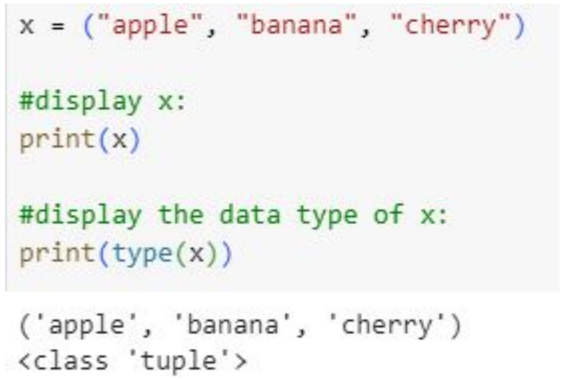
List Data Type and Tuple Datatype

Lists are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type. List are mutable



```
x = ["apple", "banana", "cherry"]

#display x:
print(x)

#display the data type of x:
print(type(x))
```

```
['apple', 'banana', 'cherry']
<class 'list'>
```



```
x = ("apple", "banana", "cherry")

#display x:
print(x)

#display the data type of x:
print(type(x))
```

```
('apple', 'banana', 'cherry')
<class 'tuple'>
```

A tuple contains a group of elements which can be same or different types. The only difference between a tuple and a list is that tuples are immutable i.e. tuples cannot be modified after it is created. It is represented by using parentheses().Tuples are used to store data which should not be modified.

Eg:- a=(10,20,-50,29.3,'Hello')

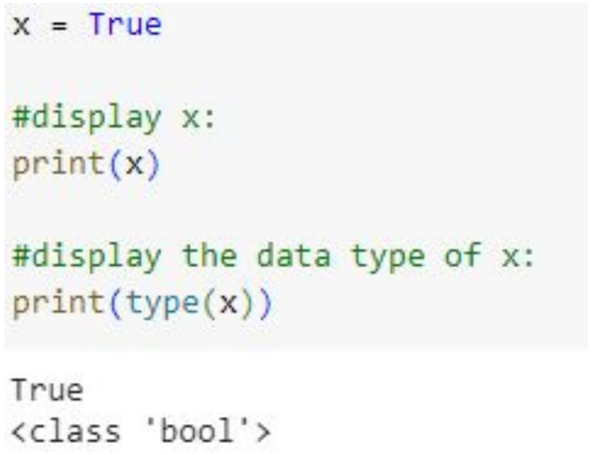
Boolean Data Type in Python

The *boolean* data type is either True or False. In Python, boolean variables are defined by the `True` and `False` keywords.

The output `<class 'bool'>` indicates the variable is a boolean data type.

Note the keywords `True` and `False` must have an Upper Case first letter.

Using a lowercase `true` returns an error.



```
x = True

#display x:
print(x)

#display the data type of x:
print(type(x))
```

```
True
<class 'bool'>
```

Set Data Type in Python

A set is unordered collection of elements much like a set in mathematics.The order of elements is not maintained in the sets.It means the elements may not appear in the same order as they are entered into the set.A set **doesnot accept duplicate** elements.Set is **immutable**.Sets are unordered so we can not access its elem**ents using index**.Sets are represented using curly brackets{}.Unordered means that the items in a set do not have a defined order.

Unordered:Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

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

Unchangeable:Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Duplicates Not Allowed:Sets cannot have two items with the same value.

Note: The values True and 1 are considered the same value in sets, and are treated as duplicates:

```
thisset = {"apple", "banana", "cherry", True, 1, 2}

print(thisset)
```

```
{True, 2, 'apple', 'cherry', 'banana'}
```

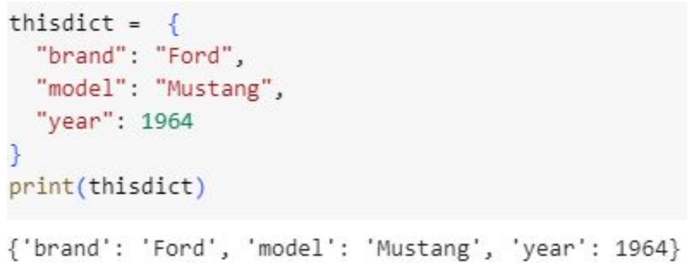
```
thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)
```

```
{'apple', 'cherry', 'banana'}
```

Dictionary Data Type in Python

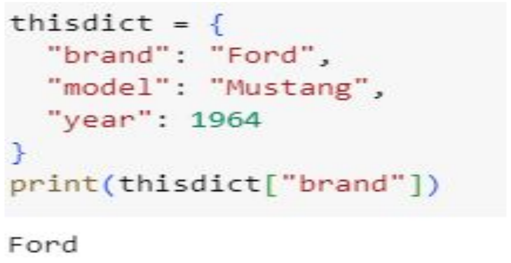
A dictionary in Python is an unordered collection of data values, used to store data values like a map, unlike other Data Types that hold only a single value as an element, a Dictionary holds a key: value pair. Key-value is provided in the dictionary to make it more optimized.

Each key-value pair in a Dictionary is separated by a colon : whereas each key is separated by a ‘comma’.Dictionaries are written with curly brackets, and have keys and values:



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

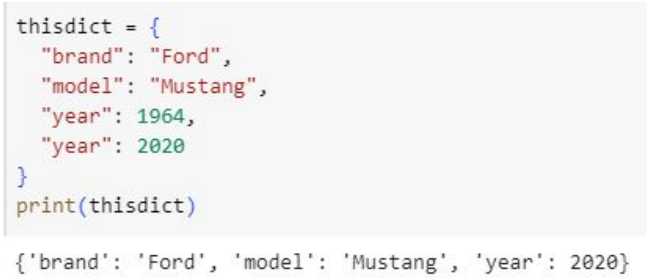
```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```



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

```
Ford
```

Dictionaries cannot have two items with the same key:



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

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Reading input in Python

This function first takes the input from the user and converts it into a string. It does not evaluate the expression it just returns the complete statement as String.

Syntax:

```
inp = input('STATEMENT')
```

How the input function works in Python :

- When input() function executes program flow will be stopped until the user has given input.
- The text or message displayed on the output screen to ask a user to enter an input value is optional i.e. the prompt, which will be printed on the screen is optional.
- Whatever you enter as input, the input function converts it into a string. if you enter an integer value still input() function converts it into a string. You need to explicitly convert it into an integer in your code using typecasting.

Print Output

In Python, we can simply use the `print()` function to print output.

Python print() function prints the message to the screen or any other standard output device.

Flow Controls

A program's `control flow` is the order in which the program's code executes.

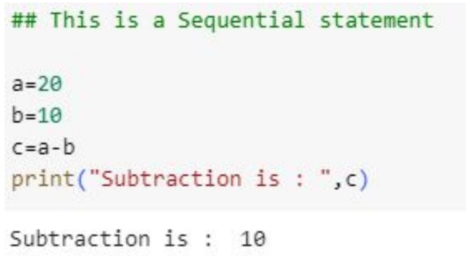
The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Python has *three* types of control structures:

- **Sequential** - default mode
- **Selection** - used for decisions and branching
- **Repetition** - used for looping, i.e., repeating a piece of code multiple times.

1. Sequential

Sequential statements are a set of statements whose execution process happens in a sequence. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.



```
## This is a Sequential statement

a=20
b=10
c=a-b
print("Subtraction is : ",c)
```

```
Subtraction is :  10
```

2. Selection/Decision control statements

In Python, the selection statements are also known as *Decision control statements* or conditional *branching statements*.

The selection statement allows a program to test several conditions and execute instructions based on which condition is true.

Some decision control statements are:

- `if`
- `if-else`
- `nested if`
- `if-elif-else`

if statement

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

`Syntax`

```
if condition:

   # Statements to execute if

     # condition is true
```

if statement

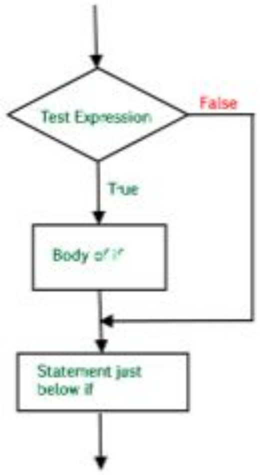
python uses indentation to identify a block. So the block under an if statement will be identified as shown in the below example:

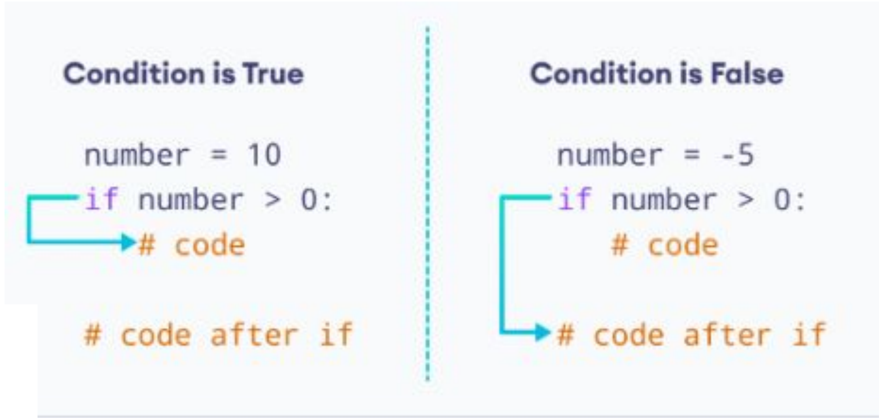
```
if condition:

   statement1

statement2
```

```
# python program to illustrate If statement

i = 10

if (i > 15):
  print("10 is less than 15")
print("I am Not in if")
```

```
n = 10
if n % 2 == 0:
   print("n=",n ,"is a even number")
```

```
n= 10 is a even number
```





```
number = 10
if number > 0:
    # code

# code after if
```

```
number = -5
if number > 0:
    # code

# code after if
```

Working of if Statement

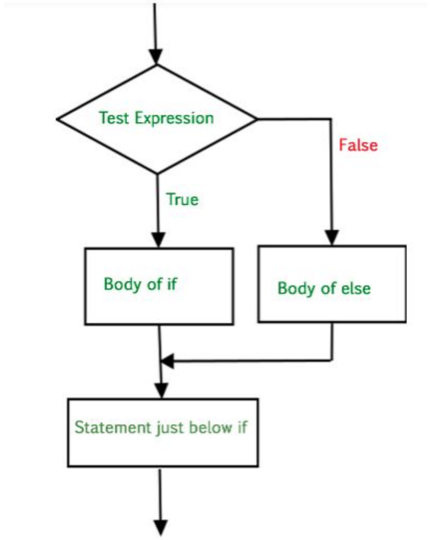
if...else statement

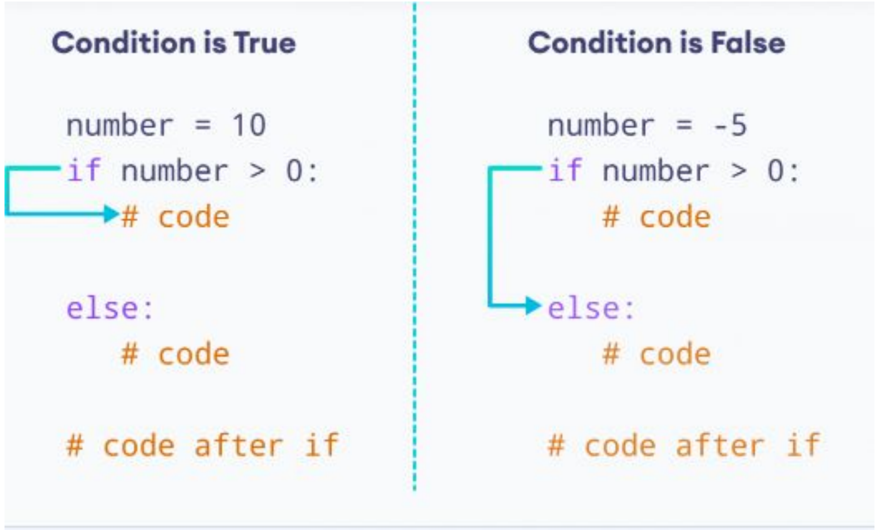
The `if-else` statement evaluates the condition and will execute the body of `if` if the test condition is `True`, but if the condition is `False`, then the body of `else` is executed.

The `if...else` statement is used to execute a block of code among two alternatives.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
      # condition is false
```



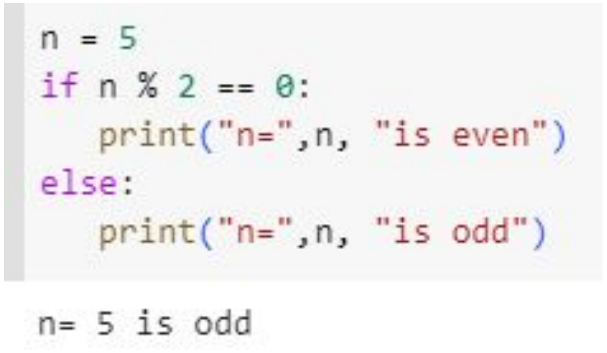


```
number = 10
if number > 0:
    # code

else:
    # code

# code after if
```

```
number = -5
if number > 0:
    # code

else:
    # code

# code after if
```

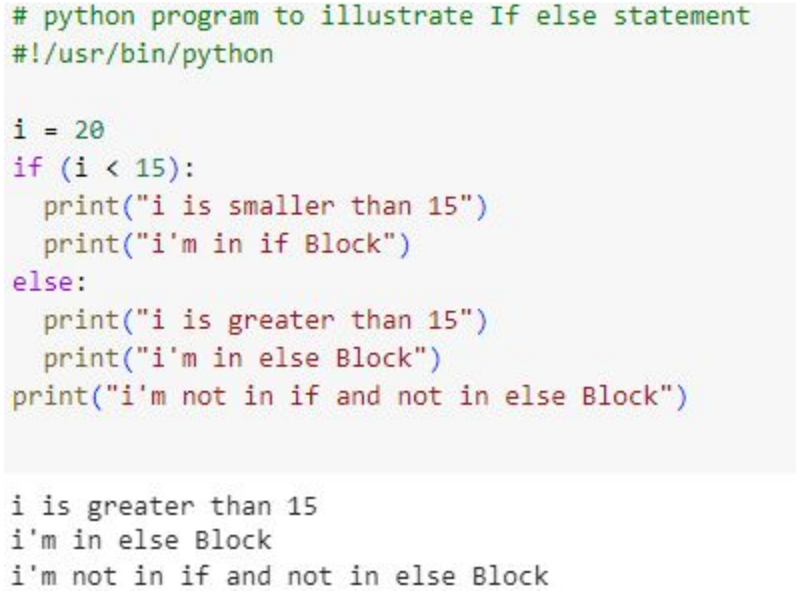
Working of if...else Statement

Examples



```python
n = 5
if n % 2 == 0:
   print("n=",n, "is even")
else:
   print("n=",n, "is odd")
```

```
n= 5 is odd
```



```python
# python program to illustrate If else statement
#!/usr/bin/python

i = 20
if (i < 15):
  print("i is smaller than 15")
  print("i'm in if Block")
else:
  print("i is greater than 15")
  print("i'm in else Block")
print("i'm not in if and not in else Block")
```

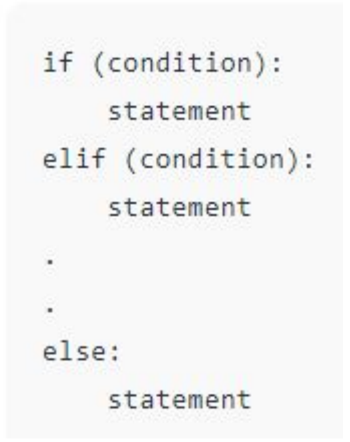
```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

The if…elif…else Decision control statements

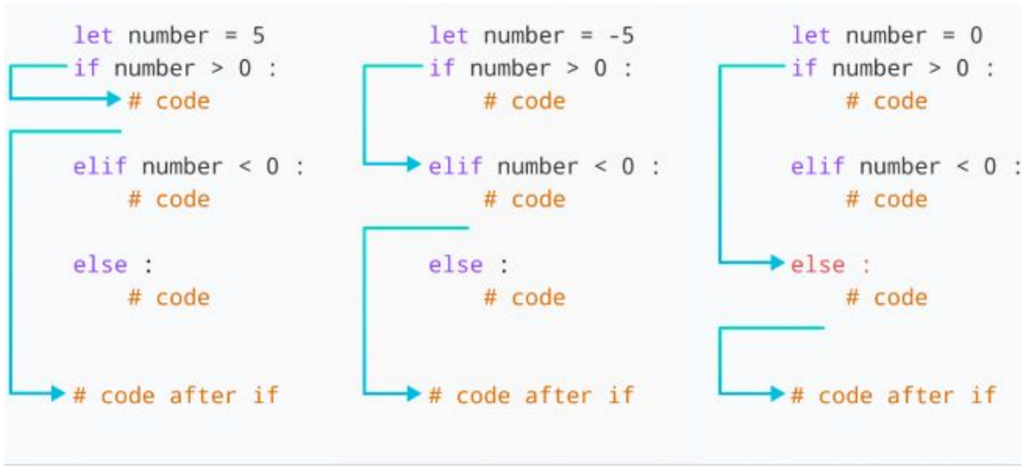
The `if-elif-else` statement is used to conditionally execute a statement or a block of statements.

However, if we need to make a choice between more than two alternatives, then we use the `if...elif...else` statement.The syntax of the `if...elif...else` statement is:

Syntax:



```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```



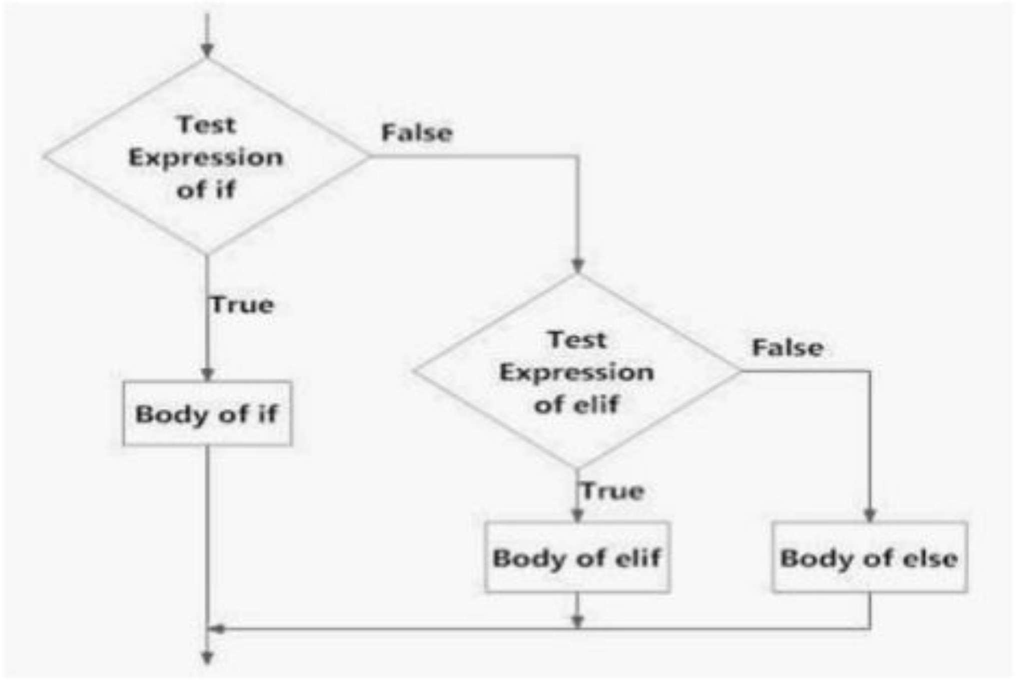
```
let number = 5
if number > 0 :
    # code

elif number < 0 :
    # code

else :
    # code

# code after if
```

```
let number = -5
if number > 0 :
    # code

elif number < 0 :
    # code

else :
    # code

# code after if
```

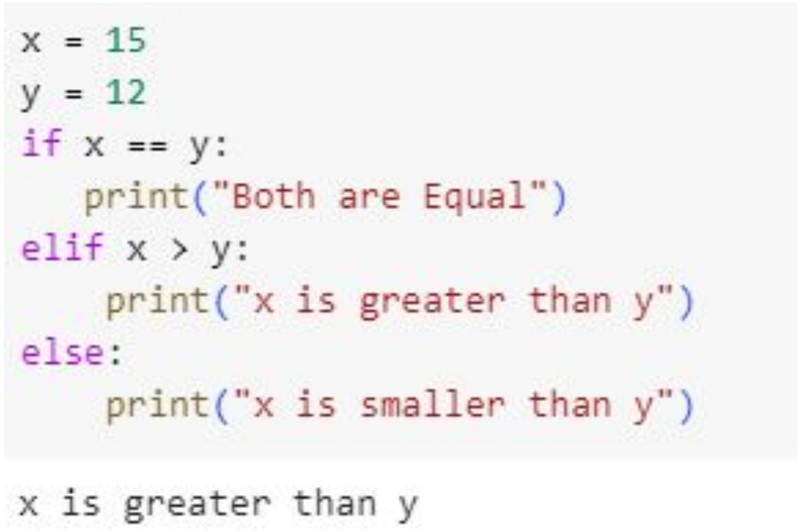
```
let number = 0
if number > 0 :
    # code

elif number < 0 :
    # code

else :
    # code

# code after if
```

Working of if...elif Statement

The if…elif…else

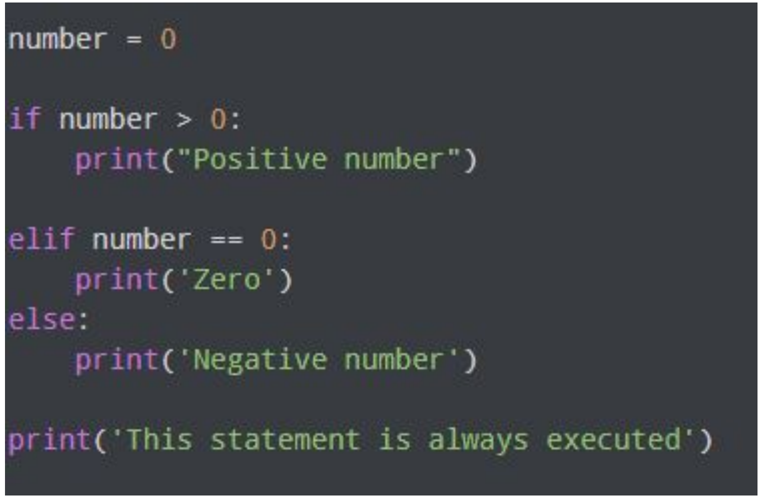


Example



```
x = 15
y = 12
if x == y:
   print("Both are Equal")
elif x > y:
    print("x is greater than y")
else:
    print("x is smaller than y")
```

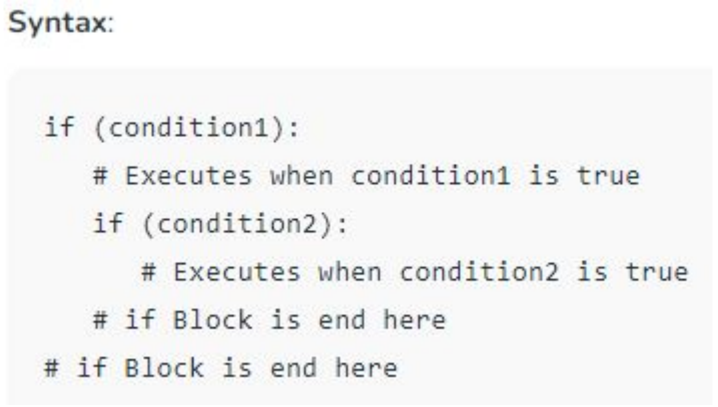
```
x is greater than y
```



```
number = 0

if number > 0:
    print("Positive number")

elif number == 0:
    print('Zero')
else:
    print('Negative number')

print('This statement is always executed')
```

Nested if statements

An `if` statement inside of an `if` statement. This is known as a nested if statement.

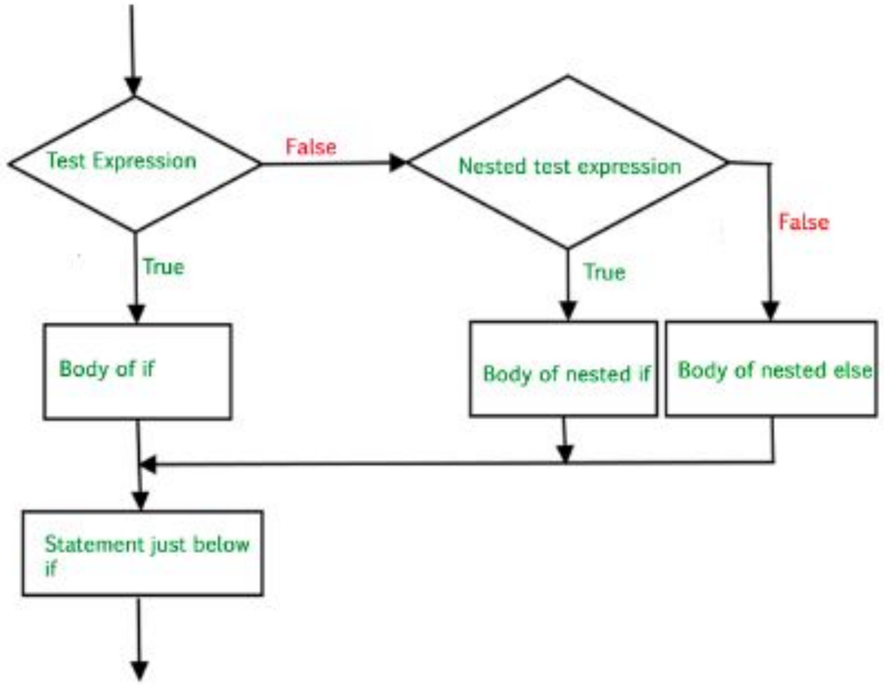


Syntax:

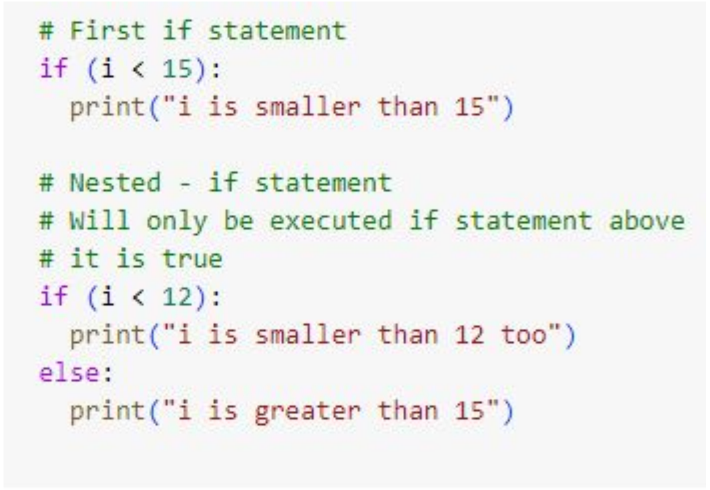
```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Task1: nested if statement to check whether the given number is positive, negative, or 0.

Nested if statements

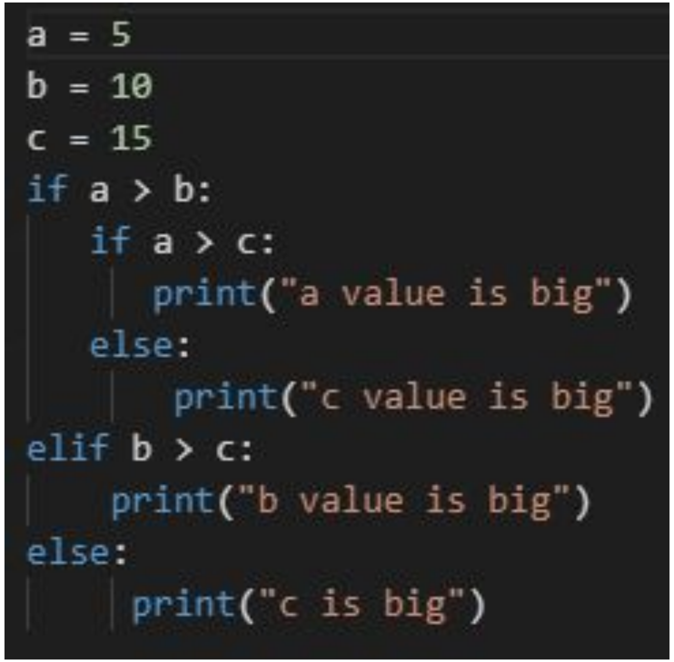


Examples



```
# First if statement
if (i < 15):
  print("i is smaller than 15")

# Nested - if statement
# Will only be executed if statement above
# it is true
if (i < 12):
  print("i is smaller than 12 too")
else:
  print("i is greater than 15")
```

```
i is smaller than 15
i is smaller than 12 too
```



```
a = 5
b = 10
c = 15
if a > b:
    if a > c:
        print("a value is big")
    else:
        print("c value is big")
elif b > c:
    print("b value is big")
else:
    print("c is big")
```

Python Loops

While Loop in Python

In python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

Loops are either infinite or conditional. Python while loop keeps reiterating a block of code defined inside it until the desired condition is met. The while loop contains a boolean expression and the code inside the loop is repeatedly executed as long as the boolean expression is true. The statements that are executed inside while can be a single line of code or a block of multiple statements.

Syntax:

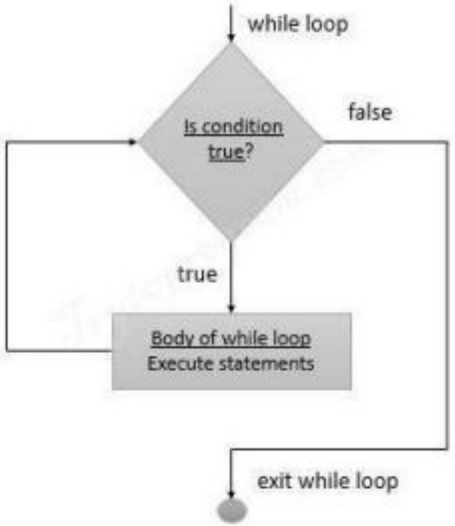
```
while expression:
    statement(s)
```

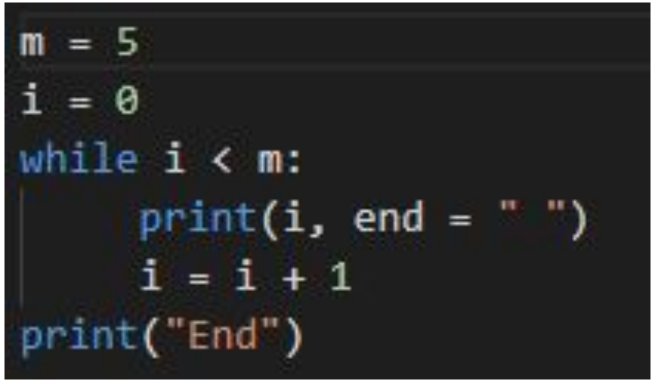
1. A `while` loop evaluates the `condition`
2. If the `condition` evaluates to `True`, the code inside the `while` loop is executed.
3. `condition` is evaluated again.
4. This process continues until the condition is `False`.
5. When `condition` evaluates to `False`, the loop stops.

```python
# Python program to illustrate while loop
count = 0
while (count < 3):
  count = count + 1
  print("Hello Geek")
```

```
Hello Geek
Hello Geek
Hello Geek
```

Flowchart:

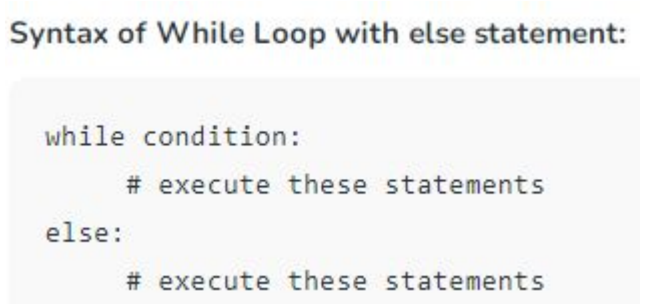




```
m = 5
i = 0
while i < m:
    print(i, end = " ")
    i = i + 1
print("End")
```

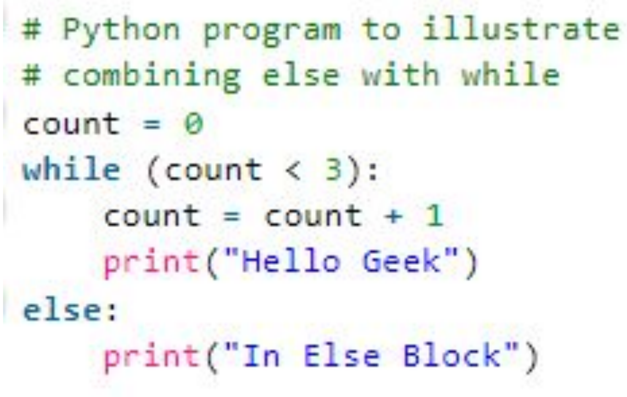
Using else statement with While Loop in Python

The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.



Syntax of While Loop with else statement:

```
while condition:
     # execute these statements
else:
     # execute these statements
```



```
# Python program to illustrate
# combining else with while
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
else:
    print("In Else Block")
```

Python For Loops

For Loop in Python

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is “for in” loop which is similar to for each loop in other languages. Let us learn how to use for in loop for sequential traversals.

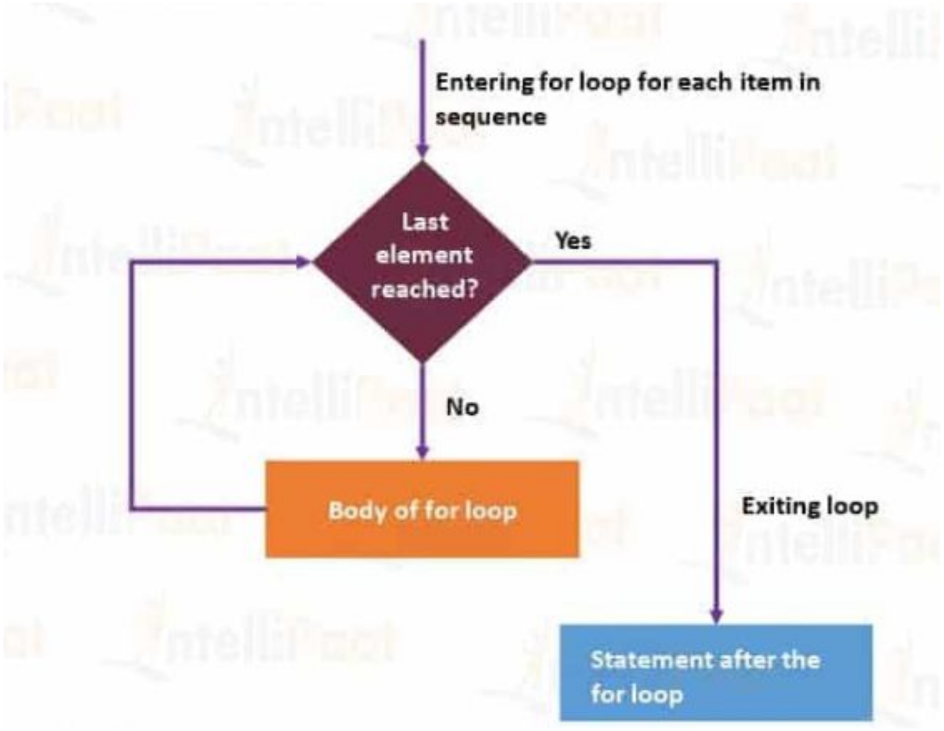
Syntax:

```
for iterator_var in sequence:
    statements(s)
```

```
# Python program to illustrate
# Iterating over range 0 to n-1

n = 4
for i in range(0, n):
  print(i)
```

```
0
1
2
3
```

Python For Loops

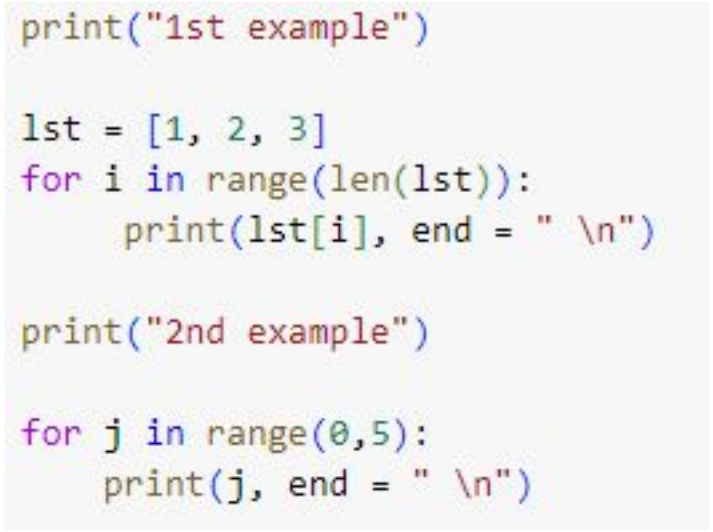

Entering for loop for each item in sequence
Last element reached?
Yes
No
Body of for loop
Exiting loop
Statement after the for loop

Example with List, Tuple, string, and dictionary iteration using For Loops in Python

We can use for loop to iterate lists, tuples, strings and sets,dictionaries in Python.

```
# Python program to illustrate
# Iterating over a list
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
  print(i)
# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("geeks", "for", "geeks")
for i in t:
  print(i)
# Iterating over a String
print("\nString Iteration")
s = "Geeks"
for i in s:
  print(i)

# Iterating over a set
print("\nSet Iteration")
set1 = {1, 2, 3, 4, 5, 6}
for i in set1:
  print(i)
```

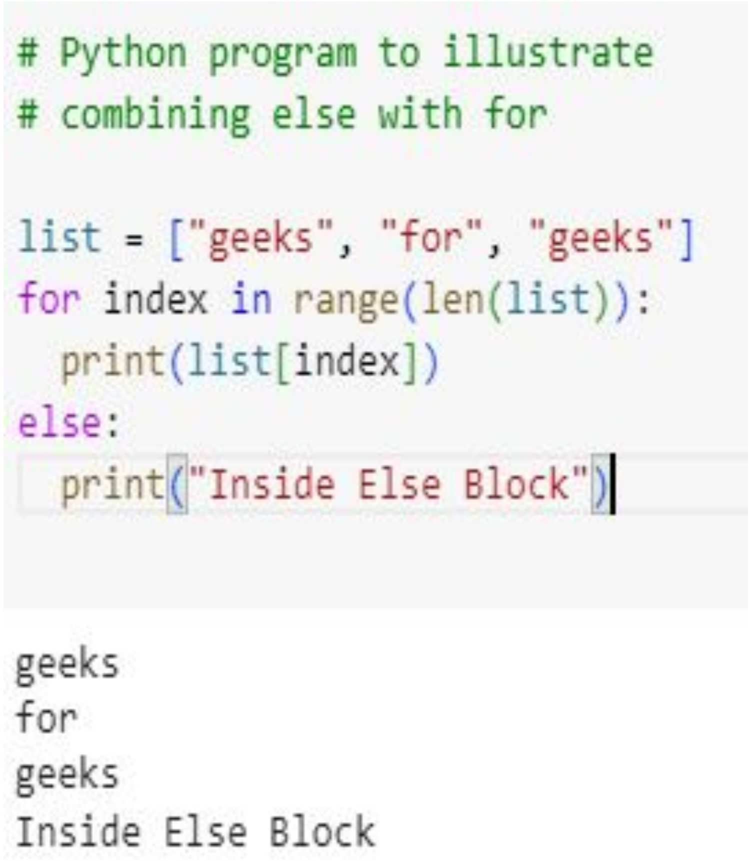
Example 2:



```
print("1st example")

lst = [1, 2, 3]
for i in range(len(lst)):
     print(lst[i], end = " \n")

print("2nd example")

for j in range(0,5):
    print(j, end = " \n")
```

Using else statement with for loop in Python

We can also combine else statement with for loop like in while loop.
But as there is no condition in for loop based on which the execution will terminate
so the else block will be executed immediately
after for block finishes execution.

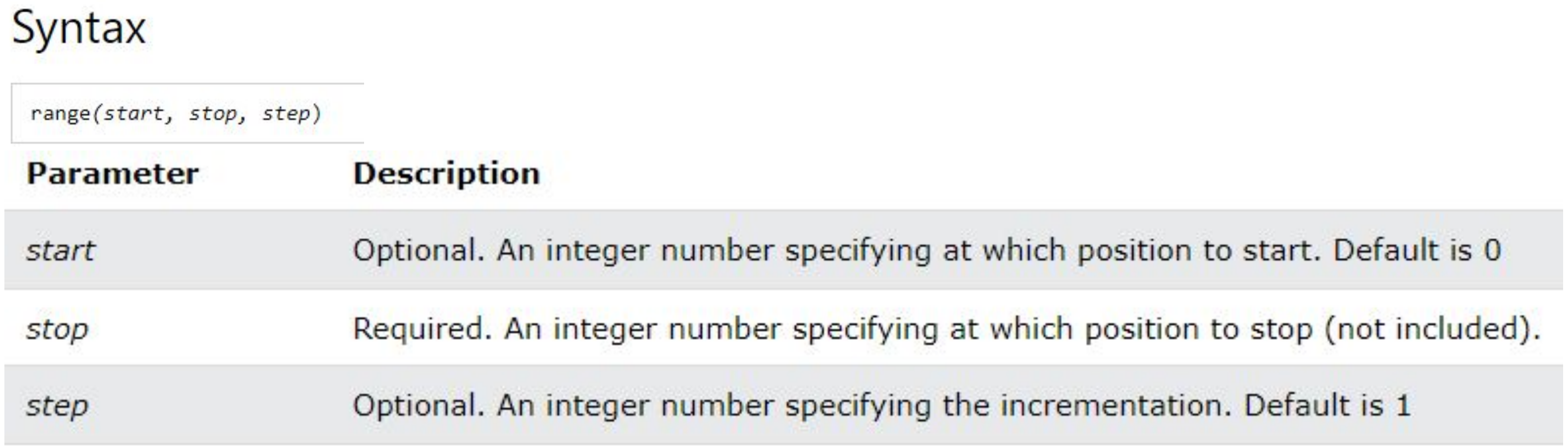


```
# Python program to illustrate
# combining else with for

list = ["geeks", "for", "geeks"]
for index in range(len(list)):
  print(list[index])
else:
  print("Inside Else Block")
```

```
geeks
for
geeks
Inside Else Block
```

range() in python

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.



Syntax

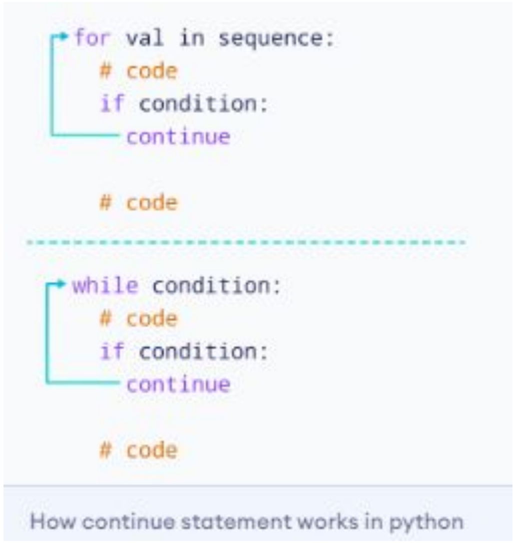
```
range(start, stop, step)
```

Parameter	Description
start	Optional. An integer number specifying at which position to start. Default is 0
stop	Required. An integer number specifying at which position to stop (not included).
step	Optional. An integer number specifying the incrementation. Default is 1

Python Continue Statement

The continue statement in Python used to skip the rest of the code inside a loop for current iteration only.Loop doesnot terminate but continue on with the next iteration.

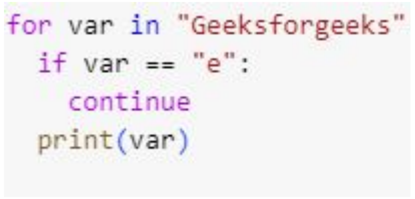
The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.



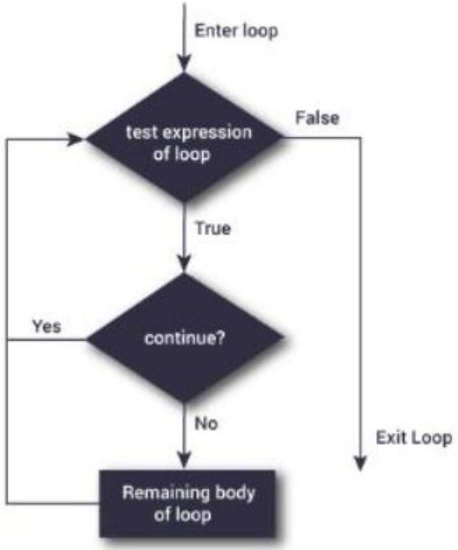
```
for val in sequence:
    # code
    if condition:
        continue

    # code
```

```
while condition:
    # code
    if condition:
        continue

    # code
```

How continue statement works in python



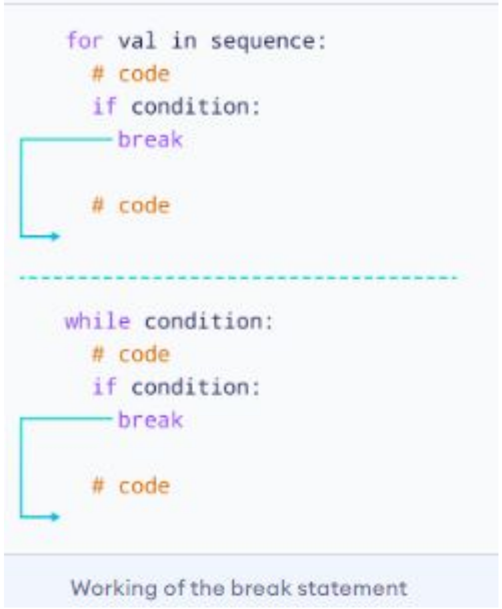
```
for var in "Geeksforgeeks"
  if var == "e":
    continue
  print(var)
```



Python break Statement

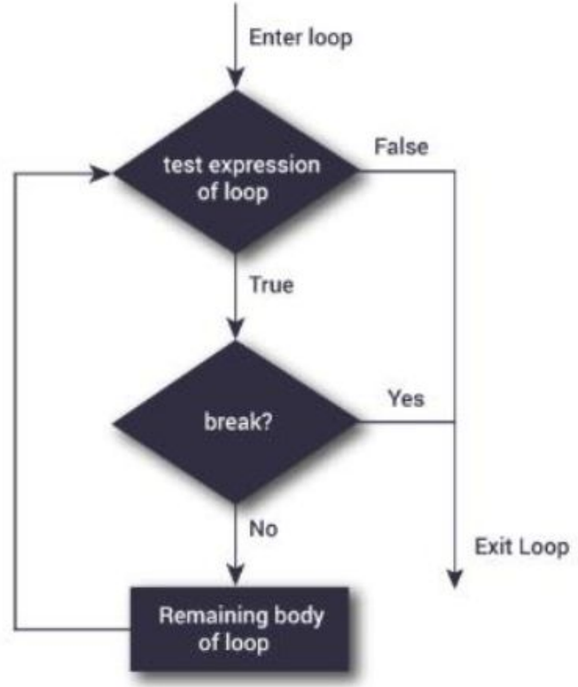
The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.The break statement can be used in both *while* and *for* loops.

```
for var in "Geeksforgeeks":
 if var == "e":
  continue
 print(var)
```



```
for val in sequence:
  # code
  if condition:
    break

  # code

-----------------------------------

while condition:
  # code
  if condition:
    break

  # code
```

Working of the break statement

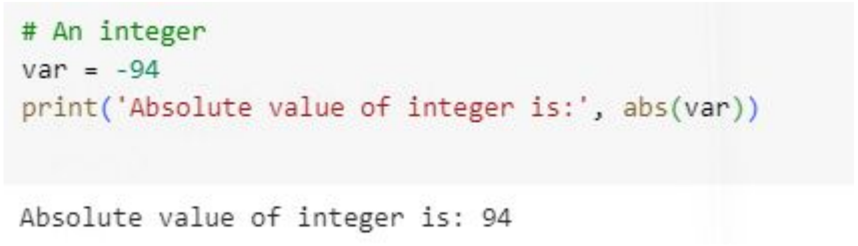


Python Built in Functions

Python provides a lot of built-in functions that ease the writing of code.

abs() in Python:- The Python abs() function return the absolute value. The absolute value of any number is always positive it removes the negative sign of a number in Python.
Syntax: *abs(number)*

- ***number:*** *Integer, floating-point number, complex number.*

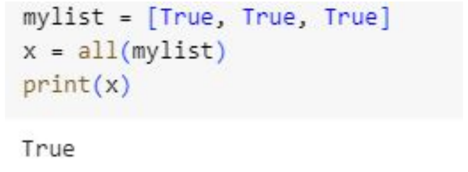


```
# An integer
var = -94
print('Absolute value of integer is:', abs(var))
```

```
Absolute value of integer is: 94
```

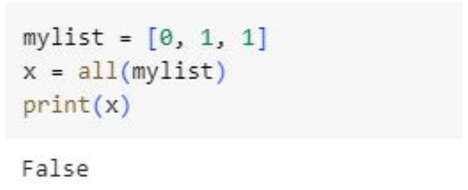
Python all() Function

The **Python all() function** returns true if all the elements of a given iterable (List, Dictionary, Tuple, set, etc.) are True otherwise it returns False.



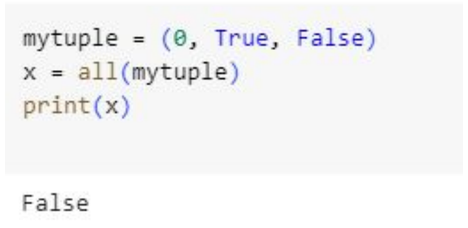
```python
mylist = [True, True, True]
x = all(mylist)
print(x)
```

```
True
```



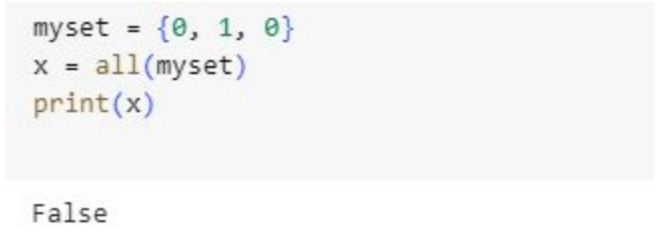
```python
mylist = [0, 1, 1]
x = all(mylist)
print(x)
```

```
False
```



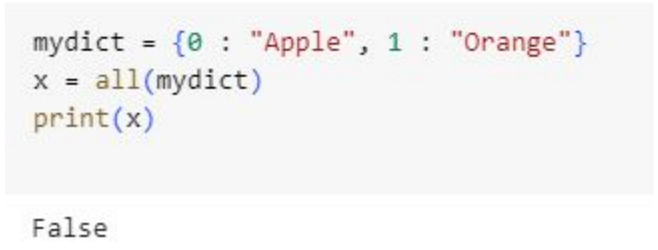
```python
mytuple = (0, True, False)
x = all(mytuple)
print(x)
```

```
False
```



```python
myset = {0, 1, 0}
x = all(myset)
print(x)
```

```
False
```



```python
mydict = {0 : "Apple", 1 : "Orange"}
x = all(mydict)
print(x)
```

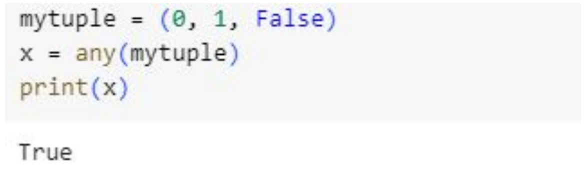
```
False
```

Python any() Function

Python any() function returns True if any of the elements of a given iterable(List, Dictionary, Tuple, set, etc) are True else it returns False.

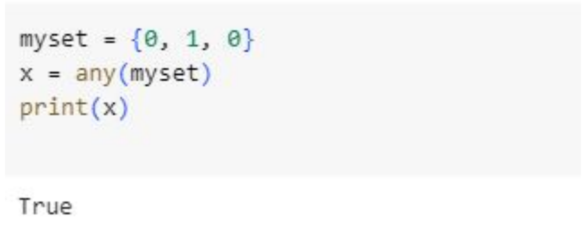
Syntax: *any(iterable)*

Iterable: *It is an iterable object such as a dictionary, tuple, list, set, etc.*



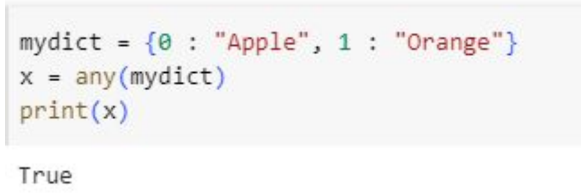
```
mytuple = (0, 1, False)
x = any(mytuple)
print(x)
```

```
True
```



```
myset = {0, 1, 0}
x = any(myset)
print(x)
```

```
True
```



```
mydict = {0 : "Apple", 1 : "Orange"}
x = any(mydict)
print(x)
```

```
True
```

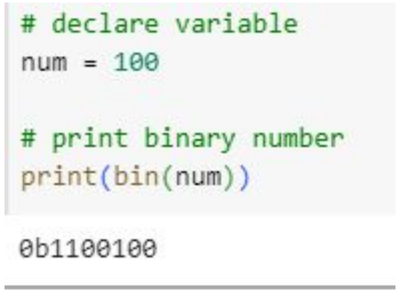
bin() in Python

Python bin() function returns the binary string of a given integer.The result will always have the prefix 0b

Synatx:

```
bin(n)
```

*n*Required. An integer



```
# declare variable
num = 100

# print binary number
print(bin(num))
```

```
0b1100100
```

Python complex() Function

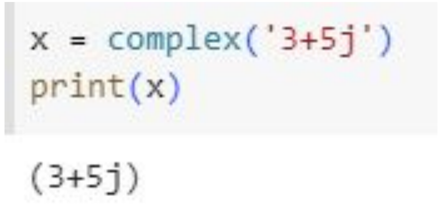
The `complex()` function returns a complex number by specifying a real number and an imaginary number.

Syntax

```
complex(real, imaginary)
```

Example

Convert a string into a complex number:



```
x = complex('3+5j')
print(x)
```

```
(3+5j)
```

Modules in Python

Why Modules?

In any project we have to write big programs.It becomes difficult to manage & organize programs

So we break down big programs into small manageable & organized files having some logic.These files are nothing but modules.

It provides reusability to our code.

Note: This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

Syntax:

```
import  <module-name>
```

What are modules in python?

- A Python module is a file containing Python definitions and statements.
- A module can define functions, classes, and variables.
- A module can also include runnable code.
- Grouping related code into a module makes the code easier to understand and use.
- It also makes the code logically organized.

Renaming the Python module

We can rename the module while importing it using the keyword.You can create an alias when you import a module, by using the `as` keyword:

Syntax: *Import* ***Module_name*** *as* ***Alias_name***

Eg:-Import addition as a

a.add(6,2)

Two types of Modules

1)User defined modules

2)Built-in modules

User defined Modules

# A simple module

addition.py

```
def add(num1, num2):
        return (num1+num2)
def subtract(num1, num2):
        return (num1+num2)
```

Calculator.py

```
Import addition
addition.add(4,5)
```

Or

```
Import addition as a
a.add(6,2)
```

```
from addition import add
  add(3,5)
or
from addition import add
  sub(3,5)
#To import all
from addition import *
  sub(4,6)
Or
from addition import add,sub
  add(6,7)
```

Built-in modules:

Here are several built-in modules in Python, which you can import whenever you like.Import and use the `platform` module:

```
Eg 1)import platform
        x = platform.system()
        print(x)
```

o/p:
Windows

Eg 2)**Python Datetime:**A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

The date contains year, month, day, hour, minute, second, and microsecond.

The `datetime` module has many methods to return information about the date object.

```
import datetime

x = datetime.datetime.now()
print(x)
```

```
o/p:2023-07-10 12:29:48.122657
```

Creating Date Objects

To create a date, we can use the `datetime()` of the `datetime` module.

The `datetime()` class requires three parameters to create a date: year, month, day.

Create a date object:

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

The `datetime()` class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of `0`, (`None` for timezone).

The Math Module

Python has also a built-in module called `math`, which extends the list of mathematical functions.

To use it, you must import the `math` module:

```
import math
```

The `math.sqrt()` method for example, returns the square root of a number

```
import math

x = math.sqrt(64)

print(x)
```

The `math.ceil()` method rounds a number upwards to its nearest integer, and the `math.floor()` method rounds a number downwards to its nearest integer, and returns the result:

```
import math

x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1
```

The `math.pi` constant, returns the value of PI (3.14...):

```
import math

x = math.pi

print(x)
```

RegEx Module

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

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

Import the `re` module:

```
import re
```

The findall() Function:The `findall()` function returns a list containing all matches.

```
import re


#Return a list containing every occurrence of "ai":


txt = "The rain in Spain"

x = re.findall("ai", txt)

print(x)
```

Return an empty list if no match was found:

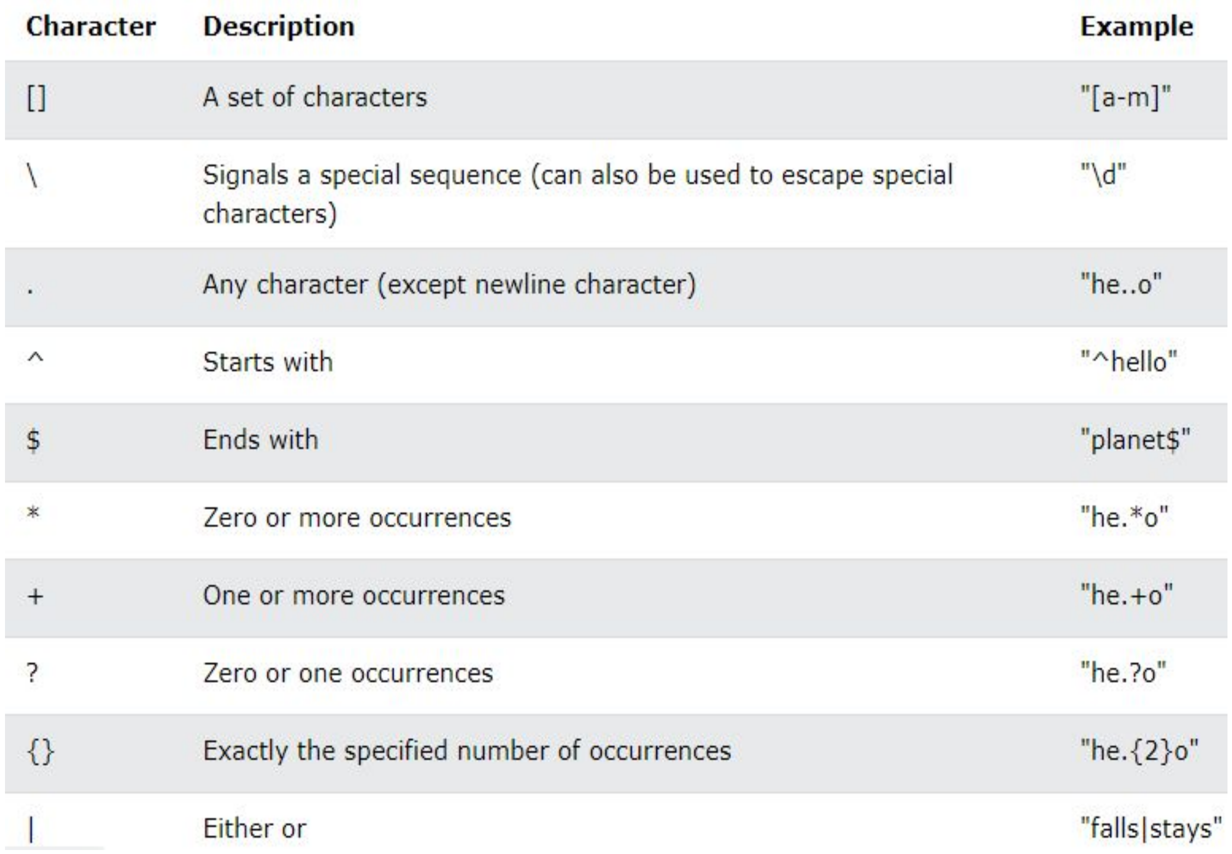
```
import re

txt = "The rain in Spain"

#Check if "Portugal" is in the string:

x = re.findall("Portugal", txt)
print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

The split() Function:

The `split()` function returns a list where the string has been split at each match:

```
import re

#Split the string at every white-space character:

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

Metacharacters

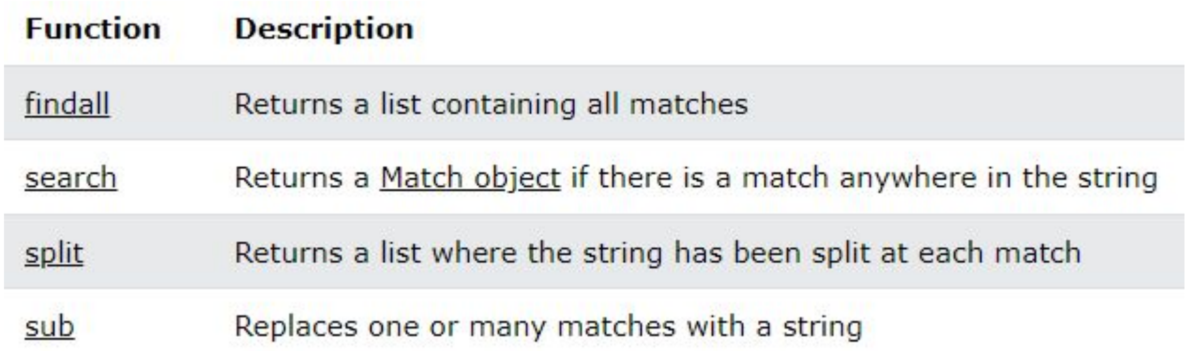


Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"

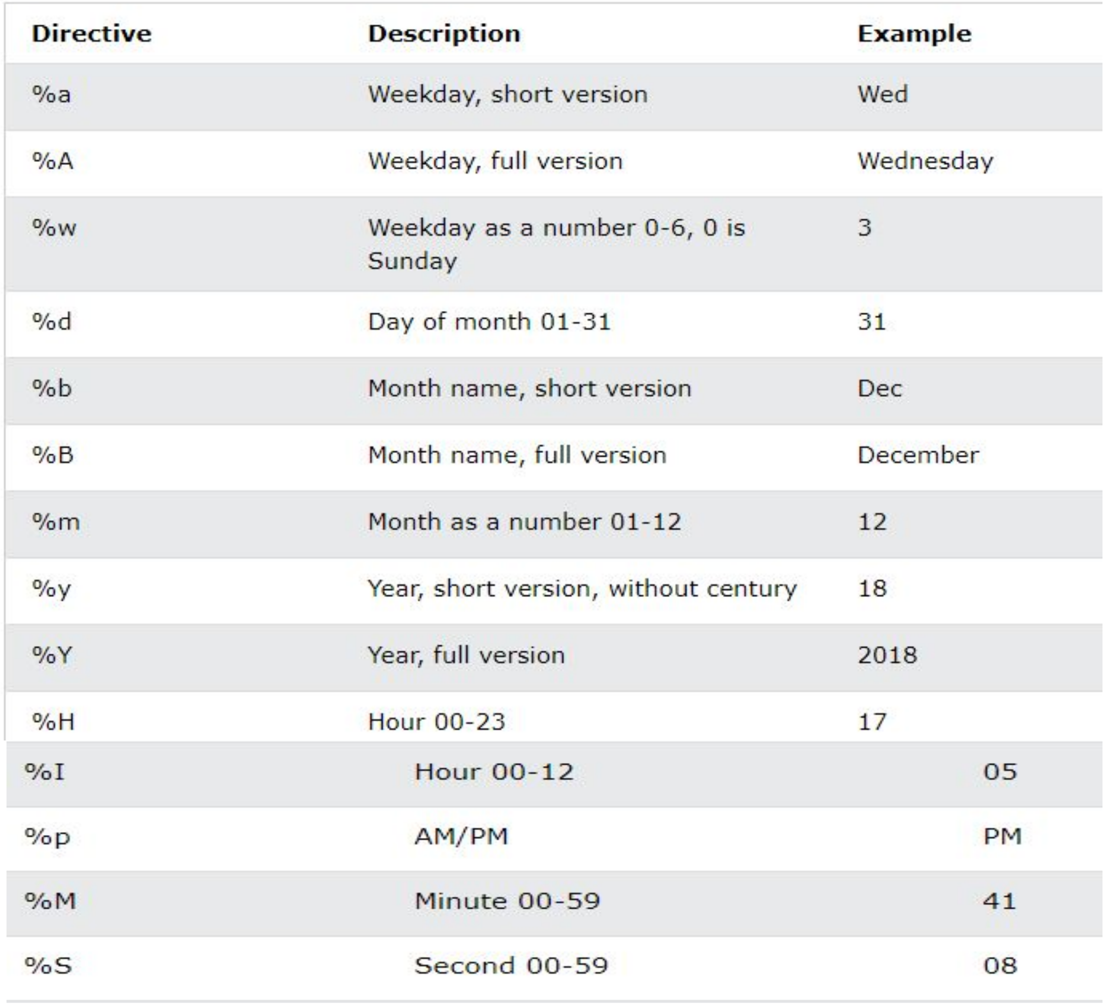
Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:



Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string



Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08

```
import re

s = 'GeeksforGeeks: A computer science portal for geeks'

match = re.search('portal', s)

print('Start Index:', match.start())
print('End Index:', match.end())
```

```
import re


s = 'geeks.forgeeks'


# without using \
match = re.search('.', s)
print(match)


# using \
match = re.search('\.', s)
print(match)
```

Functions in Python

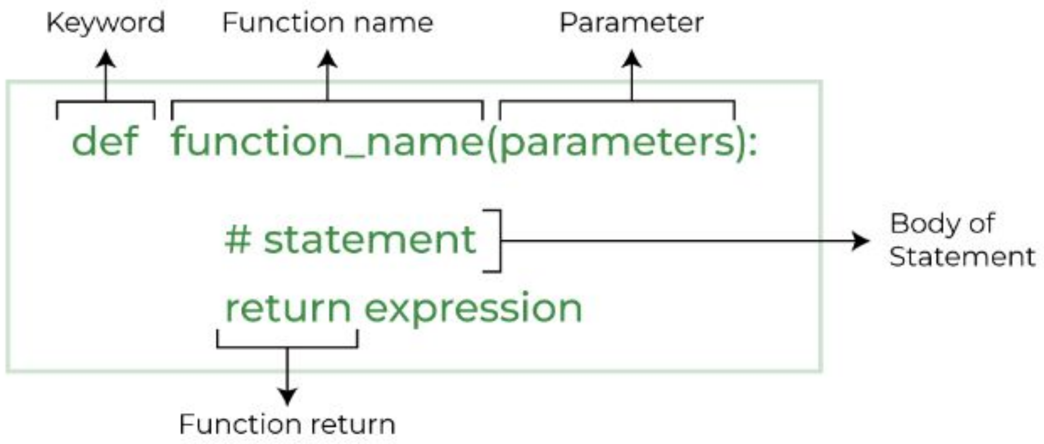
Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Some **Benefits of Using Functions**

- Increase Code Readability
- Increase Code Reusability

Python Function Declaration

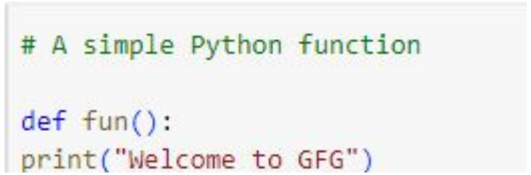
The syntax to declare a function is:



```
def function_name(parameters):
    # statement
    return expression
```

Function Definition in Python

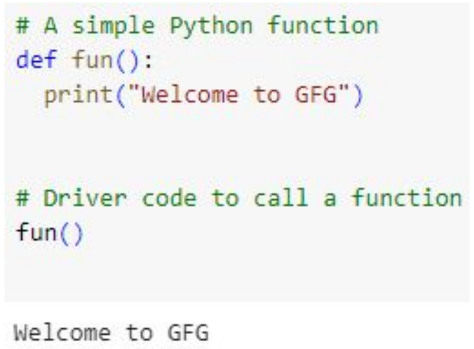
We can create a user-defined function in Python, using the **def** keyword. We can add any type of functionalities and properties to it as we require.



```python
# A simple Python function

def fun():
print("Welcome to GFG")
```

Calling the Function in Python

After creating a function in Python we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.



```python
# A simple Python function
def fun():
  print("Welcome to GFG")


# Driver code to call a function
fun()
```

```
Welcome to GFG
```

Defining and calling a function with parameters

Syntax

```
def function_name(parameter: data_type) -> return_type:

    # body of the function

       return expression
```

```
Eg:def add(num1: int, num2: int) -> int:

    """Add two numbers"""

    num3 = num1 + num2

    return num3

# Driver code

num1, num2 = 5, 15

ans = add(num1, num2)

print("The addition of”,num1 ,“and”, num2 “results” ans")
```

Python return statement

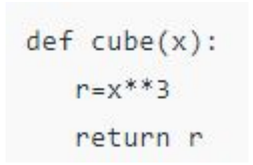
A **return statement** is used to end the execution of the function call and "returns" the result (value of the expression following the return keyword) to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned. A **return statement** is overall used to invoke a function so that the passed statements can be executed.

Note: *Return statement can not be used outside the function.*

Syntax:

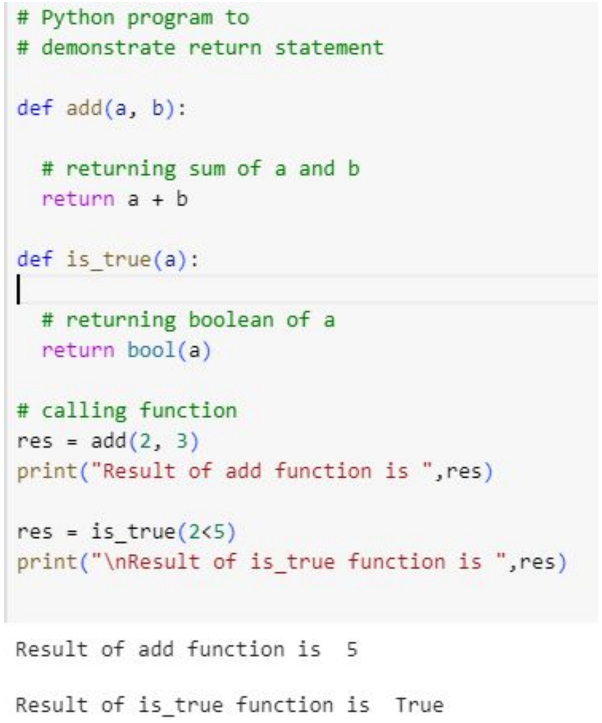
```
def fun():

    Statements

    .

    .

      return [expression]
```

Example



```
def cube(x):
    r=x**3
    return r
```

Example



```
# Python program to
# demonstrate return statement

def add(a, b):

  # returning sum of a and b
  return a + b

def is_true(a):

  # returning boolean of a
  return bool(a)

# calling function
res = add(2, 3)
print("Result of add function is ",res)

res = is_true(2<5)
print("\nResult of is_true function is ",res)
```

```
Result of add function is  5

Result of is_true function is  True
```

Pass by Object Reference in Python

In Python programming language, the mechanism is ***pass by object reference***. This technique is unique as compared to other programming languages.We know that almost everything in Python is an object. Thus, every value is an object in Python, and all values contain references to actual objects.

In other words, we can say, object references that refer to the actual objects are passed by value. We call this mechanism as pass by object reference in Python.

Key Points about Pass by Object Reference

There are the following important points about the pass by object reference you should keep it mind.

- In the Python programming language, all argument values are passed to a function by object reference.
- If the object is mutable (changeable), the changed value is available outside the function. Mutable objects may be list, set, dictionary, byte array.
- If the object is immutable (not changeable), the changed value is not available outside the function. Immutable objects may be numbers (int, float, complex), strings, tuple, frozen set, bytes.
- If we pass immutable objects to the function, the passing works like pass by value or call by value.
- If we pass mutable objects to the function, the passing works like pass by reference or call by reference.

Example 1:

Let’s write a Python program in which we will pass mutable objects (list) to the function using pass by object reference.

```
# This program illustrates the concept of call by object reference using mutable objects.

def my_funct(y):
    print('Values inside the function before modifying:',y)
    y[2] = 4 # modifying value.
    print('Values inside the function after modifying:',y)

# Main part of program.
x = [1, 2, 3] # List of values.
print('Values outside the function before modifying:',x)

my_funct(x) # calling function with passing x to function parameter y.
print('Values outside the function after modifying:',x)
```

```
def modify(x):
   x=15
   print(x, id(x))
x= 10
modify(x)
print(x, id(x))
```

In this example, ‘x’ is a list of values. Since the list is a mutable object in Python, the passing acts like call by reference. We have passed x as an argument to the function’s parameter y using call by reference.

Here, y is a formal parameter, whereas x is an actual argument. Both x and y belong to the same memory location. If we make any change in y inside the function, then it also affects in x outside the function.

In this scenario, Python makes only one copy of the argument in the memory location for processing. Values are the same in that copy, but parameter names are different, x and y.

Example 2:

Let’s write a Python program in which we will pass immutable objects (string) to the function using pass by object reference.

```
# This program illustrates the concept of call by object reference using immutable objects.
def my_func(name):
    print('String inside the function before modifying:',name)
    name = 'John' # modifying value.
    print('String inside the function after modifying:',name)

# Main part of program.
n = 'Herry'
print('String outside the function before modifying:',n)

my_func(n)
# calling function with passing n to function parameter name.
print('String outside the function after modifying:',n)
```

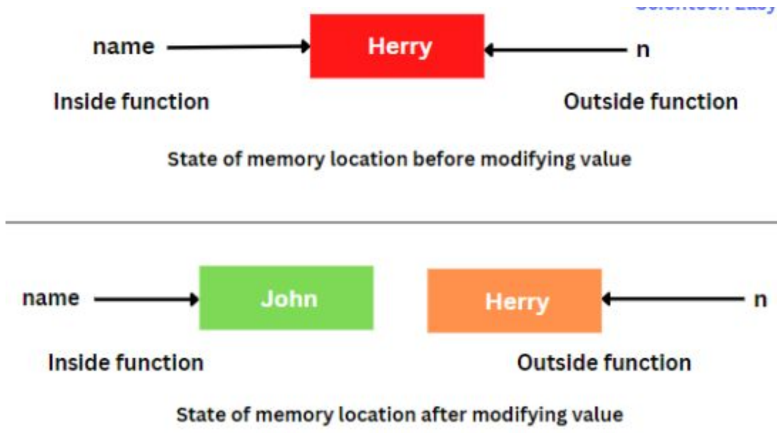

name
Herry
n
Inside function
Outside function
State of memory location before modifying value
name
John
Herry
n
Inside function
Outside function
State of memory location after modifying value

Fig: Python passes argument values to the function using the pass by reference

In this example, n is a string. Since string is an immutable object in Python, the passing acts like call by value. We have passed ‘n’ as an argument value to the function’s parameter name.

Here, ‘name’ is a formal parameter, whereas ‘n’ is an actual argument. Initially, both have the same memory location. As we have made an update in the value of name (name = ‘John’), the value of name also has updated.

In this scenario, the value of ‘n’ remains the same and the memory location of ‘n’ also remains the same. But, the memory location of ‘name’ is changed. Hence, Python makes two copies of arguments in two memory locations for processing.

Void Function in Python

Void function in Python is a function that performs an action but does not return any computed or final value to the caller.

It can accept any number of parameters and perform operation inside the function. A void function may or may not return a value to the caller.

If we do not declare return statement to return a value, Python will send a value None that represents nothingness.

The following example code demonstrates the void function, which will calculate the sum of two numbers and prints the result on the console.

# Python program demonstrates void function.

```
def voidOperation():
    num1 = int(input('Enter your first number: '))
    num2 = int(input('Enter your second number: '))
    sum = num1 + num2
    print(“Sum of”,num1,  num2, “is”,sum)
def main_fn():
    voidOperation() # calling one function from another function.
main_fn() # calling main function.
```

Parameters or Arguments?

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Python Function Arguments

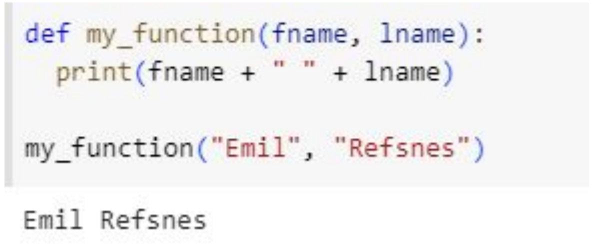
Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Number of Arguments

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

Example

This function expects 2 arguments, and gets 2 arguments:



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

my_function("Emil", "Refsnes")
```

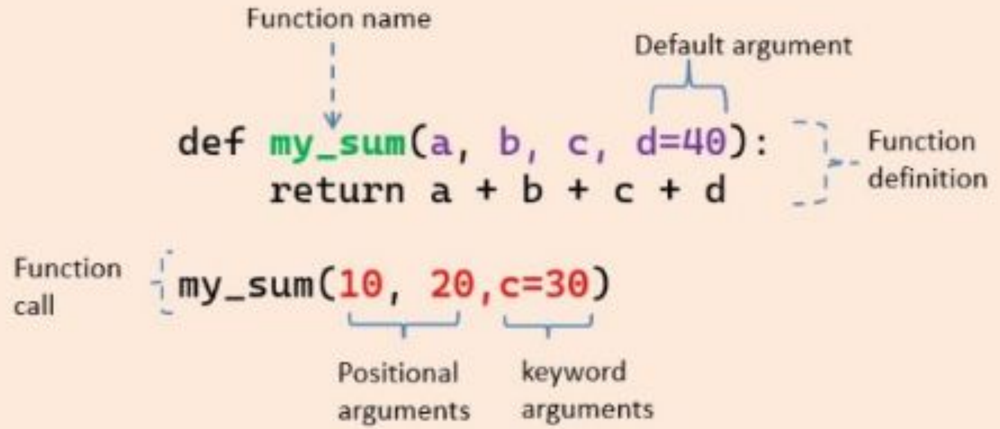
```
Emil Refsnes
```

Types of Arguments

Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

- **Default argument**
- **Keyword arguments (named arguments)**
- **Arbitrary arguments** (variable-length arguments *args and **kwargs)
- **Command Line arguments**



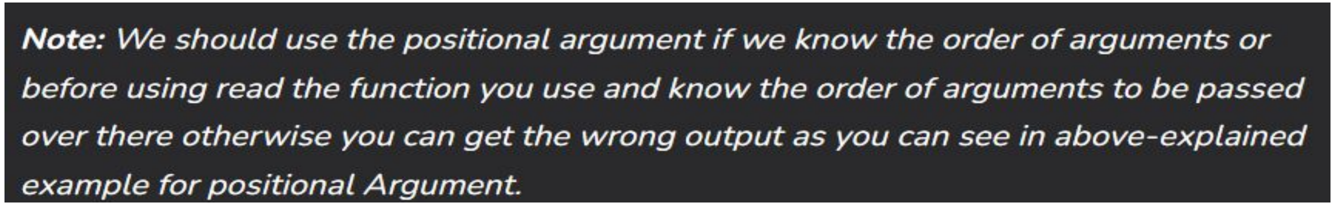
```
def my_sum(a, b, c, d=40):
    return a + b + c + d

my_sum(10, 20,c=30)
```

- **Positional argument** values get assigned as per the sequence. Now a=10 and b=20
- **Keyword arguments** are those arguments where values get assigned to the arguments by their keyword
- **Default arguments**: Assign default values to the argument using the '=' operator at the time of function definition

Positional Arguments

Position-only arguments mean whenever we pass the arguments in the order we have defined function parameters in which if you change the argument position then you may get the unexpected output. We should use positional Arguments whenever we know the order of argument to be passed. So now, we will call the function by using the position-only arguments in two ways and In both cases, we will be getting different outputs from which one will be correct and another one will be incorrect.

```
def nameAge(name, age):
        print("Hi, I am", name)
        print("My age is ", age)

# You will get correct output because argument is given in order
print("Case-1:")
nameAge("Prince", 20)
# You will get incorrect output because argument is not in order
print("\nCase-2:")
nameAge(20, "Prince")
```



Note: *We should use the positional argument if we know the order of arguments or before using read the function you use and know the order of arguments to be passed over there otherwise you can get the wrong output as you can see in above-explained example for positional Argument.*

Keyword Arguments

Keyword arguments are those arguments where **values get assigned to the arguments by their keyword (name)** when the function is called. It is preceded by the variable name and an (=) assignment operator. The Keyword Argument is also called a named argument.**Change the sequence of keyword arguments:**you can change the sequence of keyword arguments by using their name in function calls.

Python allows functions to be called using keyword arguments. But all the keyword arguments should match the parameters in the function definition. When we call functions in this way, the order (position) of the arguments can be changed.

```
# function with 2 keyword arguments

def student(name, age):
    print('Student Details:', name, age)

# default function call
student('Jessa', 14)

# both keyword arguments
student(name='Jon', age=12)

# 1 positional and 1 keyword
student('Donald', age=13)
```

- keyword arguments should follow positional arguments only.
- we can mix positional arguments with keyword arguments during a function call. But, a keyword argument must always be after a non-keyword argument (positional argument). Else, you’ll get an error.
- I.e., avoid using keyword argument before positional argument.
- The order of keyword arguments is not important, but All the keyword arguments passed must match one of the arguments accepted by the function.
- No argument should receive a value more than once

Eg:-def get_student(name, age, grade):

```
   print(name, age, grade)

get_student(name='Jessa', 12, 'Six')
```

Eg:-def get_student(name, age, grade):

```
   print(name, age, grade)

get_student(grade='Six', name='Jessa', age=12)


get_student(name='Jessa', age=12, standard='Six')
```

Eg:-def student(name, age, grade):

```
   print('Student Details:', name, grade, age)

student(name='Jon', age=12, grade='Six', age=12)
```

Default Arguments

In a function, arguments can have default values. We assign default values to the argument using the ‘=’ (assignment) operator at the time of function definition. You can define a function with any number of default arguments.

The default value of an argument will be used inside a function if we do not pass a value to that argument at the time of the function call. Due to this, the default arguments become optional during the function call.

It overrides the default value if we provide a value to the default arguments during function calls.

```
# function with 2 keyword arguments grade and school

def student(name, age, grade="Five", school="ABC School"):
    print('Student Details:', name, age, grade, school)

# without passing grade and school
# Passing only the mandatory arguments
student('Jon', 12)
```

Example:

Let's define a function student() with four arguments name, age, grade, and school. In this function, grade and school are default arguments with default values.

- If you call a function without school and grade arguments, then the default values of grade and school are used.
- The age and name parameters do not have default values and are required (mandatory) during a function call.

```
# function with 2 keyword arguments grade and school

def student(name, age, grade="Five", school="ABC School"):
    print('Student Details:', name, age, grade, school)

# without passing grade and school
# Passing only the mandatory arguments
student('Jon', 12)
```

Passing one of the default arguments:

If you pass values of the `grade` and `school` arguments while calling a function, then those values are used instead of default values.

```
# function with 2 keyword arguments grade and school
def student(name, age, grade="Five", school="ABC School"):
    print('Student Details:', name, age, grade, school)

# not passing a school value (default value is used)
# six is assigned to grade
student('Kelly', 12, 'Six')

# passign all arguments
student('Jessa', 12, 'Seven', 'XYZ School')
```

Note:Default arguments must follow the non-default argument

Variable-length arguments/Arbitrary Arguments

In Python, sometimes, there is a situation where we need to **pass multiple arguments to the function**. Such types of arguments are called arbitrary arguments or variable-length arguments.

We **use variable-length arguments if we don't know the number of arguments needed for the function in advance**.

Types of Arbitrary Arguments:

- arbitrary positional arguments (`*args`)
- arbitrary keyword arguments (`**kwargs`)

The *args and **kwargs allow you to pass multiple positional arguments or keyword arguments to a function.

Arbitrary positional arguments (`*args`)

The special syntax **args* in function definitions in Python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.

- The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
- What **args* allows you to do is take in more arguments than the number of formal arguments that you previously defined. With **args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
- For example, we want to make a multiply function that takes any number of arguments and is able to multiply them all together. It can be done using *args.
- Using the *, the variable that we associate with the * becomes iterable meaning you can do things like iterate over it, run some higher-order functions such as map and filter, etc.

Example:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Example 1:Python program to illustrate *args for a variable number of arguments

```
def myFun(*argv):
	for arg in argv:
		print(arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Example 2:Python program to illustrate *args with a first extra argument

```
def myFun(arg1, *argv):
	print("First argument :", arg1)
	for arg in argv:
		print("Next argument through *argv :", arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Arbitrary keyword arguments (**kwargs)

The special syntax ***kwargs* in function definitions in Python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is that the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.

Example 1: Python program to illustrate *kwargs for a variable number of keyword arguments

```
def myFun(**kwargs):
        for key, value in kwargs.items():
                print (key, value)

myFun(first='Geeks', mid='for', last='Geeks')
```

Example 2:Python program to illustrate **kwargs for a variable number of keyword arguments with one extra argument

```
def myFun(arg1, **kwargs):
        for key, value in kwargs.items():
                print (key, value)

myFun("Hi", first='Geeks', mid='for', last='Geeks')
```

Using both *args and **kwargs in Python to call a function

```
def myFun(arg1, arg2, arg3):
        print("arg1:", arg1)
        print("arg2:", arg2)
        print("arg3:", arg3)


# Now we can use *args or **kwargs to
# pass arguments to this function :
args = ("Geeks", "for", "Geeks")
myFun(*args)

kwargs = {"arg1": "Geeks", "arg2": "for", "arg3": "Geeks"}
myFun(**kwargs)
```

Example 2:

Here, we are passing *args and **kwargs as an argument in the myFun function. where **‘geeks’, ‘for’, ‘geeks’** is passed as *args, and **first=”Geeks”, mid=”for”, last=”Geeks”** is passed as **kwargs and printing **in the same line.**

```
def myFun(*args, **kwargs):
        print("args: ", args)
        print("kwargs: ", kwargs)


# Now we can use both *args ,**kwargs
# to pass arguments to this function :
myFun('geeks', 'for', 'geeks', first="Geeks", mid="for", last="Geeks")
```

Command line arguments

The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**.Python Command Line Arguments provides a convenient way to accept some information at the command line while running the program

For example -

```
$ python script.py arg1 arg2 arg3
```

Using sys.argv

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
One such variable is sys.argv which is a simple list structure. It’s main purpose are:

- It is a list of command line arguments.
- len(sys.argv) provides the number of command line arguments.
- sys.argv[0] is the name of the current Python script.

Here Python script name is test.py and rest of the three arguments - arg1 arg2 arg3 are command line arguments for the program.

Example 1:-test.py

```
import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'
print 'Argument List:', str(sys.argv)

To run:$ python test.py arg1 arg2 arg3
```

# Python program to demonstrate command line arguments

gfg.py

```
import sys

# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)

# Arguments passed
print("\nName of Python script:", sys.argv[0])

print("\nArguments passed:", end = " ")
for i in range(1, n):
        print(sys.argv[i], end = " ")

# Addition of numbers
Sum = 0
# Using argparse module
for i in range(1, n):
        Sum += int(sys.argv[i])

print("\n\nResult:", Sum)
```

To run

```
Py gfg.py 1 2 3 4
```

Scope and lifetime of variable

Python Scope variable

The location where we can find a variable and also access it if required is called the scope of a variable.

Python Local variable

Local variables are those that are initialized within a function and are unique to that function. It cannot be accessed outside of the function.

```
def f():

        # local variable
        s = "Geeksforgeeks"
        print("Inside Function:", s)

f()
print(s)
```

Python Global variables

Global variables are the ones that are defined and declared outside any function and are not specified to any function. They can be used by any part of the program.

```
# This function uses global variable s

def f():
        print(s)


# Global scope
s = "Geeksforgeeks"
f()
```

Global and Local Variables with the Same Name

```
# This function has a variable with
# name same as s.
def f():
        s = "Me too."
        print(s)

# Global scope
s = "Geeksforgeeks"
f()
print(s)
```

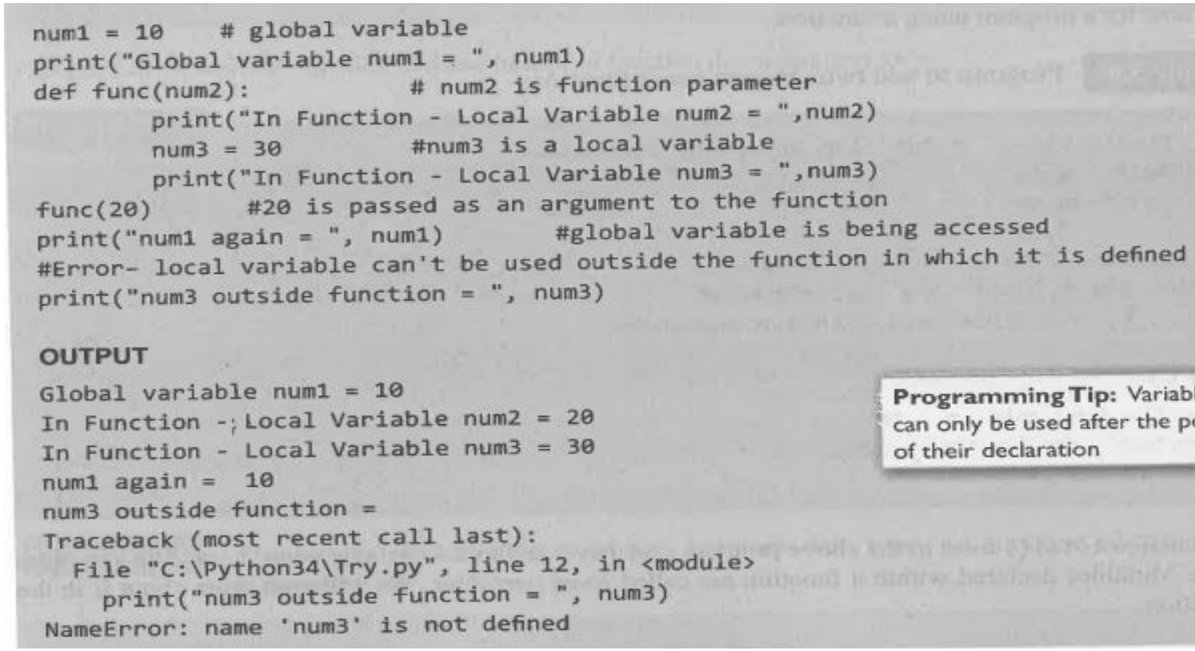
The variable s is defined as the string “Geeksforgeeks”, before we call the function f(). The only statement in f() is the print(s) statement. As there are no locals, the value from the global s will be used.

```
def f():
        print(s)


        print(s)


# Global scope
s = "I love Geeksforgeeks"
f()
print(s)
```

Using global statement

To define a variable inside a function as global ,use the global statement

```
# This function modifies global variable 's'

def f():
        global s
        print(s)
        s = "Look for Geeksforgeeks
             Python Section"
        print(s)


# Global Scope
s = "Python is great !"
f()
print(s)
```

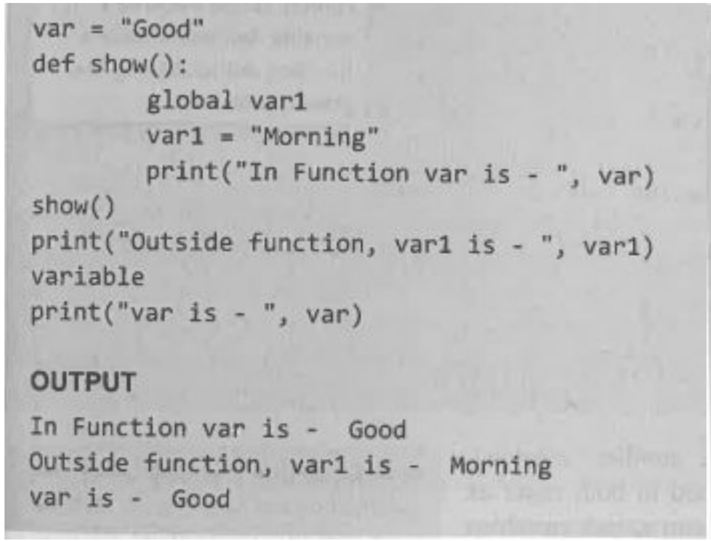


```
num1 = 10    # global variable
print("Global variable num1 = ", num1)
def func(num2):             # num2 is function parameter
        print("In Function - Local Variable num2 = ",num2)
        num3 = 30           #num3 is a local variable
        print("In Function - Local Variable num3 = ",num3)
func(20)       #20 is passed as an argument to the function
print("num1 again = ", num1)       #global variable is being accessed
#Error- local variable can't be used outside the function in which it is defined
print("num3 outside function = ", num3)
```

OUTPUT

```
Global variable num1 = 10
In Function - Local Variable num2 = 20
In Function - Local Variable num3 = 30
num1 again =  10
num3 outside function =
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 12, in <module>
    print("num3 outside function = ", num3)
NameError: name 'num3' is not defined
```

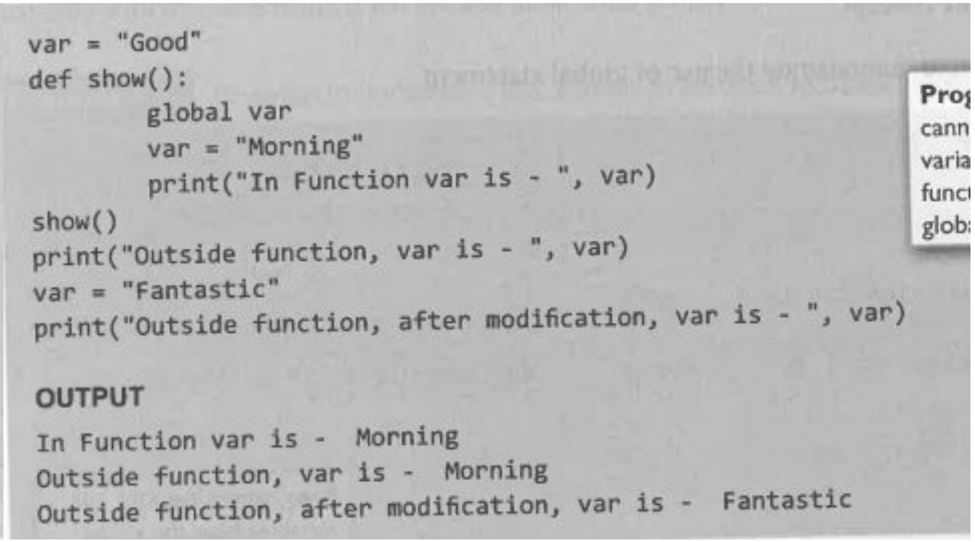
Programming Tip: Variab
can only be used after the p
of their declaration



```
var = "Good"
def show():
        global var1
        var1 = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var1 is - ", var1)
variable
print("var is - ", var)
```

OUTPUT

```
In Function var is -  Good
Outside function, var1 is -  Morning
var is -  Good
```

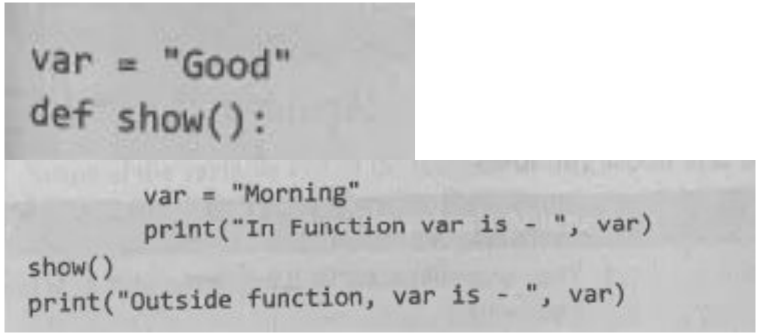


```
var = "Good"
def show():
        global var
        var = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var is - ", var)
var = "Fantastic"
print("Outside function, after modification, var is - ", var)
```

OUTPUT

```
In Function var is -  Morning
Outside function, var is -  Morning
Outside function, after modification, var is -  Fantastic
```

Prog
cann
varia
funct
glob



```
var = "Good"
def show():
        var = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var is - ", var)
```