**Identifiers:**

The identifier is a name used to identify a variable, function, class, module, etc. The identifier is a combination of character digits and underscore. The identifier should start with a character or Underscore then use a digit. The characters are A-Z or a-z, an UnderScore ( \_ ) , and digit (0-9). we should not use special characters ( #, @, $, %, ! ) in identifiers.

**Rules :**

A variable name must start with a letter or the underscore character.

* A variable name cannot start with a number.
* A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ ).
* Variable names are case-sensitive (name, Name and NAME are three different variables).
* The reserved words(keywords) cannot be used naming the variable.

**Examples of valid identifiers:**

var1

\_var1

\_1\_var

var\_1

**Examples of invalid identifiers:**

!var1

1var

1\_var

var#1

**Variables:**

Python is not “statically typed”. We do not need to declare variables before using them or declare their type. **A variable is created the moment we first assign a value to it. A variable is a name given to a memory location. It is the basic unit of storage in a program.**

* **The value stored in a variable can be changed during program execution.**
* **A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.**

**Let’s see the simple variable creation:**

# An integer assignment

age = 45

# A floating point

salary = 1456.8

# A string

name = "John"

print(age)

print(salary)

print(name)

**Output:**

45

1456.8

John

**Expressions:**

An expression is a combination of operators and operands that is interpreted to produce some other value. In any programming language, an expression is evaluated as per the precedence of its operators. So that if there is more than one operator in an expression, their precedence decides which operation will be performed first. We have many different types of expressions in Python. Let’s discuss all types along with some exemplar codes :

**1. Constant Expressions:**These are the expressions that have constant values only.

**Example:**

* Python3

|  |
| --- |
| # Constant Expressions  x **=** 15 **+** 1.3    print(x) |

**Output**

16.3

**2. Arithmetic Expressions:**An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis. The result of this type of expression is also a numeric value. The operators used in these expressions are arithmetic operators like addition, subtraction, etc. Here are some arithmetic operators in Python:

|  |  |  |
| --- | --- | --- |
| **Operators** | **Syntax** | **Functioning** |
| + | x + y | Addition |
| – | x – y | Subtraction |
| \* | x \* y | Multiplication |
| / | x / y | Division |
| // | x // y | Quotient |
| % | x % y | Remainder |
| \*\* | x \*\* y | Exponentiation |

**Example:**

Let’s see an exemplar code of arithmetic expressions in Python :

* Python3

|  |
| --- |
| # Arithmetic Expressions  x **=** 40  y **=** 12    add **=** x **+** y  sub **=** x **-** y  pro **=** x **\*** y  div **=** x **/** y    print(add)  print(sub)  print(pro)  print(div) |

**Output**

52

28

480

3.3333333333333335

**3. Integral Expressions:**These are the kind of expressions that produce only integer results after all computations and type conversions.

**Example:**

* Python3

|  |
| --- |
| # Integral Expressions  a **=** 13  b **=** 12.0    c **=** a **+** int(b)  print(c) |

**Output**

25

**4. Floating Expressions:**These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.

**Example:**

* Python3

|  |
| --- |
| # Floating Expressions  a **=** 13  b **=** 5    c **=** a **/** b  print(c) |

**Output**

2.6

**5. Relational Expressions:**In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < , >= , <=). Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end. These expressions are also called Boolean expressions.

**Example:**

* Python3

|  |
| --- |
| # Relational Expressions  a **=** 21  b **=** 13  c **=** 40  d **=** 37    p **=** (a **+** b) >**=** (c **-** d)  print(p) |

**Output**

True

**6. Logical Expressions:**These are kinds of expressions that result in either *True*or *False.*It basically specifies one or more conditions. For example, (10 == 9) is a condition if 10 is equal to 9. As we know it is not correct, so it will return False. Studying logical expressions, we also come across some logical operators which can be seen in logical expressions most often. Here are some logical operators in Python:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Syntax** | **Functioning** |
| And | P and Q | It returns true if both P and Q are true otherwise returns false |
| Or | P or Q | It returns true if at least one of P and Q is true |
| Not | not P | It returns true if condition P is false |

**Example:**

Let’s have a look at an exemplar code :

* Python3

|  |
| --- |
| P **=** (10 **==** 9)  Q **=** (7 > 5)    # Logical Expressions  R **=** P **and** Q  S **=** P **or** Q  T **=** **not** P    print(R)  print(S)  print(T) |

**Output**

False

True

True

**7. Bitwise Expressions:**These are the kind of expressions in which computations are performed at bit level.

**Example:**

* Python3

|  |
| --- |
| # Bitwise Expressions  a **=** 12    x **=** a >> 2  y **=** a << 1    print(x, y) |

**Output**

3 24

**8. Combinational Expressions:**We can also use different types of expressions in a single expression, and that will be termed as combinational expressions.

**Example:**

* Python3

|  |
| --- |
| # Combinational Expressions  a **=** 16  b **=** 12    c **=** a **+** (b >> 1)  print(c) |

**Output**

22

**Precedence and Associativity of Operators:**

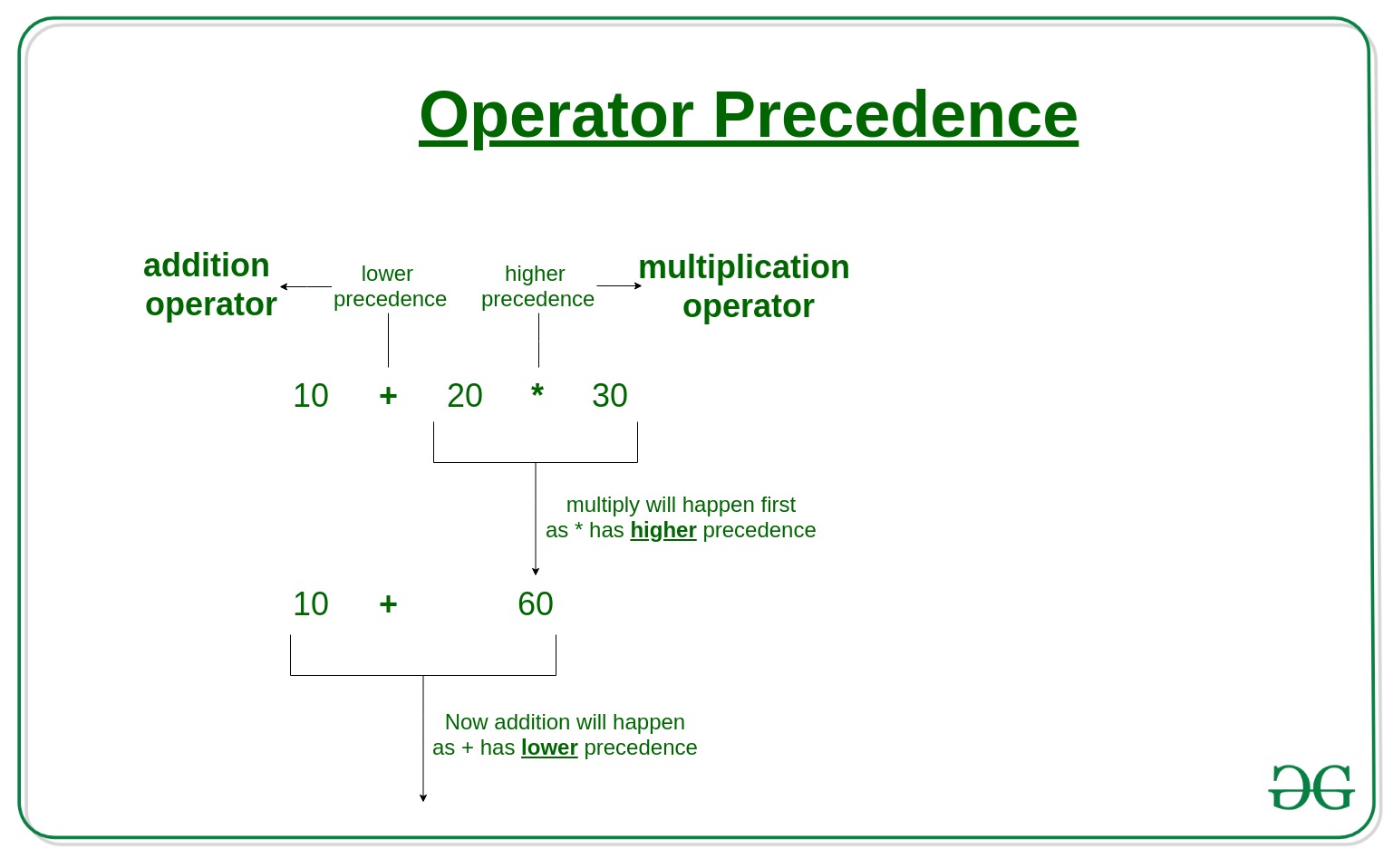
Precedence and Associativity of Operators: Operator precedence and associativity determine the priorities of the operator.

When dealing with operators in Python we have to know about the concept of Python operator precedence and associativity as these determine the priorities of the operator otherwise, we’ll see unexpected outputs.

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

**Example:** Solve

10 + 20 \* 30

[](https://media.geeksforgeeks.org/wp-content/uploads/20190708163349/Operators-Precedence.jpg)

**10 + 20 \* 30** is calculated as **10 + (20 \* 30)**

and not as **(10 + 20) \* 30**

**Code:**

* Python3

|  |
| --- |
| # Precedence of '+' & '\*'  expr **=** 10 **+** 20 **\*** 30    print(expr) |

**Operator Precedence:**

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

**Example:** Solve

100 + 200 / 10 - 3 \* 10

100 + 200 / 10 - 3 \* 10 is calculated as

100 + (200 / 10) - (3 \* 10)

and not as (100 + 200) / (10 - 3) \* 10

**Code:**

* Python3

|  |
| --- |
| expression  **=** 100 **+** 200 **/** 10 **-** 3 **\*** 10  print(expression ) |

**Output:**

90.0

Please see the following precedence and associativity table for reference. This table lists all operators from the highest precedence to the lowest precedence.

**Operator Associativity:**

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine.

It can either be **L**eft to **R**ight or from **R**ight to **L**eft.

**Example:** ‘\*’ and ‘/’ have the same precedence and their associativity is **L**eft**t**o **R**ight, so the expression “100 / 10 \* 10” is treated as “(100 / 10) \* 10”.

**Code:**

* Python3

|  |
| --- |
| # Left-right associativity  # 100 / 10 \* 10 is calculated as  # (100 / 10) \* 10 and not  # as 100 / (10 \* 10)  print(100 **/** 10 **\*** 10)    # Left-right associativity  # 5 - 2 + 3 is calculated as  # (5 - 2) + 3 and not  # as 5 - (2 + 3)  print(5 **-** 2 **+** 3)    # left-right associativity  print(5 **-** (2 **+** 3))    # right-left associativity  # 2 \*\* 3 \*\* 2 is calculated as  # 2 \*\* (3 \*\* 2) and not  # as (2 \*\* 3) \*\* 2  **print**(2 **\*\*** 3 **\*\*** 2) |

**Output:**

100

6

0

512

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | # Multi-operator expression    a **=** 10 **+** 3 **\*** 4  print(a)    b **=** (10 **+** 3) **\*** 4  **print**(b)    c **=** 10 **+** (3 **\*** 4)  print(c) |   **Output**  22  52  22  **Operator** | **Description** | **Associativity** |
| ( ) | Parentheses | left-to-right |
| \*\* | Exponent | right-to-left |
| \*  /  % | Multiplication/division/modulus | left-to-right |
| +  – | Addition/subtraction | left-to-right |
| <<  >> | Bitwise shift left, Bitwise shift right | left-to-right |
| <  <=  >  >= | Relational less than/less than or equal to  Relational greater than/greater  than or equal to | left-to-right |
| ==  != | Relational is equal to/is not equal to | left-to-right |
| is,  is not  in, not in | Identity  Membership operators | left-to-right |
| & | Bitwise AND | left-to-right |
| ^ | Bitwise exclusive OR | left-to-right |
| | | Bitwise inclusive OR | left-to-right |
| not | Logical NOT | right-to-left |
| and | Logical AND | left-to-right |
| or | Logical OR | left-to-right |
| =  +=  -=  \*=  /=  %=  &=  ^=  |=  <<=  >>= | Assignment  Addition/subtraction assignment  Multiplication/division assignment  Modulus/bitwise AND assignment  Bitwise exclusive/inclusive OR assignment  Bitwise shift left/right assignment | right-to-left |

**Multiple operators in expression (Operator Precedence):**

| **Precedence** | **Name** | **Operator** |
| --- | --- | --- |
| 1 | Parenthesis | ( ) [ ] { } |
| 2 | Exponentiation | \*\* |
| 3 | Unary plus or minus, complement | -a , +a , ~a |
| 4 | Multiply, Divide, Modulo | /  \*  //  % |
| 5 | Addition & Subtraction | +  – |
| 6 | Shift Operators | >>  << |
| 7 | Bitwise AND | & |
| 8 | Bitwise XOR | ^ |
| 9 | Bitwise OR | | |
| 10 | Comparison Operators | >=  <=  >  < |
| 11 | Equality Operators | ==  != |
| 12 | Assignment Operators | =  +=  -=  /=  \*= |
| 13 | Identity and membership operators | is, is not, in, not in |
| 14 | Logical Operators | and, or, not |

So, if we have more than one operator in an expression, it is evaluated as per operator precedence. For example, if we have the expression “10 + 3 \* 4”. Going without precedence it could have given two different outputs 22 or 52. But now looking at operator precedence, it must yield 22. Let’s discuss this with the help of a Python program:

# Multi-operator expression

a = 10 + 3 \* 4

print(a)

b = (10 + 3) \* 4

print(b)

c = 10 + (3 \* 4)

print(c)

**Output**

22

52

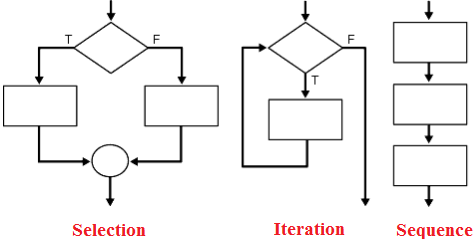
22

**Control Structures:**

According to the structure theorem, any computer program can be written using the basic **control structures** . A control structure (or flow of control) is a block of programming that analyses variables and chooses a direction in which to go based on given parameters. In simple sentence, **a control structure is just a decision that the computer makes. So, it is the basic decision-making process in programming** and **flow of control determines how a computer program will respond when given certain conditions and parameters.**

There are **two basic aspects** of computer programming: **data** and **instructions** . To work with data, you need to understand variables and data types; to work with instructions, you need to understand control structures and statements.

**Flow of control** through any given program is implemented with **3 basic types of control structures: Sequential, Selection and Repetition.**



## **Sequential:**

## Sequential execution is when statements are executed one after another in order. You don't need to do anything more for this to happen.

## **Selection:**

## Selection used for decisions, branching - choosing between 2 or more alternative paths.

* if
* if...else
* switch

## **Repetition:**

Repetition used for looping, i.e. repeating a piece of code multiple times in a row.

* while loop
* do..while loop
* for loop

These **control structures** can be combined in computer programming. A sequence may contain several loops; a loop may contain a loop nested within it, or the two branches of a conditional may each contain sequences with loops and more conditionals. From the following lessons you can understand the control structures and statements in Python language.

# **Python Conditional Statements:**

Decision making is one of the most important concepts of **computer programming** . It require that the developer specify one or more **conditions** to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. **Python** programming language provides following types of decision making statements.

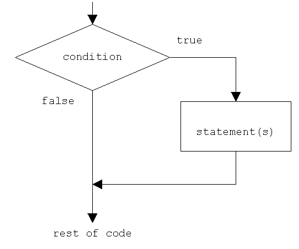
* if statements
* if....else statements
* if..elif..else statements
* nested if statements

## **if statements:**

Syntax:

if expression :

statements



In Python, if statement evaluates the test expression inside parenthesis. If test expression is evaluated to true (nonzero) , statements inside the body of if is executed. If test expression is evaluated to false (0) , statements inside the body of if is skipped.

**Example:**

x=20

y=10

if x > y :

print(" X is bigger ")

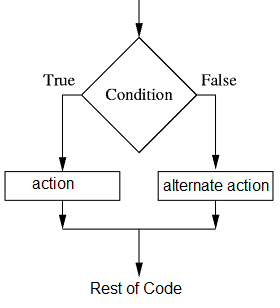
**Output:**

X is bigger

In this program we have two variables x and y. x is assigned as the value 20 and y is 10. In next line, the if statement evaluate the expression (x>y) is true or false. In this case the x > y is true because x=20 and y=10, then the control goes to the body of if block and print the message "X is bigger". If the condition is false then the control goes outside the if block.

**if..else statements:**

The else statement is to specify a block of code to be executed, if the condition in the if statement is false. Thus, the else clause ensures that a sequence of statements is executed.



**Syntax:**

**if expression :**

**statements**

**else:**

**statements**

**Example:­­**

x=int(input(“enter x value”)

y= int(input(“enter y value”)

if x > y :

print(" X is bigger ")

else :

print(" Y is bigger ")

**Output:**

Y is bigger

# If the number is positive, we print an appropriate message

num=int(input("enter num value"))

if num > 0:

print(num, "is a positive number.")

else:

print(num, "is a Neagtive number.")

In the above code, the if stat evaluate the expression is true or false. In this case the x > y is false, then the control goes to the body of else block , so the program will execute the code inside else block.

## **if..elif..else statements:**

**Syntax:**

if expression:

statements

elif expression:

statements

else:

statements

The elif is short for else if and is useful to avoid excessive indentation.

#### **Example:**

x=500

if x > 500 :

print(" X is greater than 500 ")

elif x < 500 :

print(" X is less than 500 ")

elif x == 500 :

print(" X is 500 ")

else :

print(" X is not a number ")

**Output:**

X is 500

In the above case Python evaluates each expression one by one and if a true condition is found the statement(s) block under that expression will be executed. If no true condition is found the statement(s) block under else will be executed.

## **Nested if statements:**

## In some situations you have to place an if statement inside another statement.

**Syntax:**

if condition:

if condition:

statements

else:

statements

else:

statements

**Example:**

mark = 72

if mark > 50:

if mark > = 80:

print ("You got A Grade !!")

elif mark > =60 and mark < 80 :

print ("You got B Grade !!")

else:

print ("You got C Grade !!")

else:

print("You failed!!")

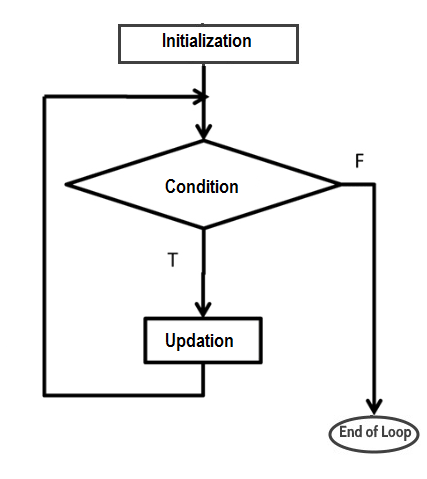
**Output:**

You got B Grade !!

**Python Iterative/Repetition Statements:**

Loops are one of the most important features in computer **programming languages** . As the name suggests is the process that get **repeated again and again** . It offer a quick and easy way to do something repeated until a certain condition is reached. Every loop has 3 parts:

* Initialization
* Condition
* Updation



## **while loop:**

**Syntax:**

**while (condition) :**

**statement(s)**

In Python, **while loop** is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. That means, while loop tells the computer to do something as long as the condition is met. It consists of **condition/expression** and a block of code. The condition/expression is evaluated, and if the condition/expression is true, the code within the block is executed. This repeats until the condition/expression becomes false.

**Syntax:**

initialization;

while(condition)

{

//Code block to execute something

}

For example, if I **initialize** the value of a variable x as 0 and set the condition x < =5 then the **condition** will be held true. But if I set the condition x>=5 the condition will become false. After checking the condition in while clause, if it holds true, the body of the loop is executed. While executing the body of loop it can update the statement inside **while loop** . After updating, the condition is checked again. This process is repeated as long as the condition is true and once the condition becomes false the program breaks out of the loop.

**Example:**

x=0

while(x < =5):

print(x)

x+=1

**Output:**

0

1

2

3

4

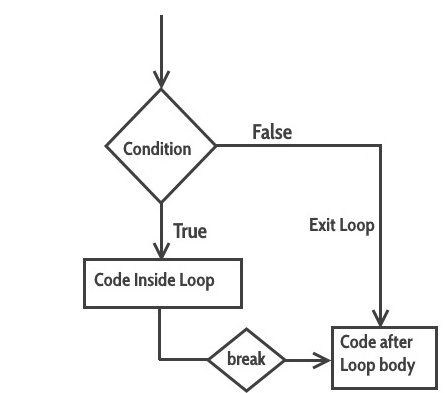
5

Here the conditional of x < =5 (while(x < =5):) and x was previously declared and set equal to 0 (x=0). So, the first item printed out was 0 (print(x)), which makes sense. In the next line x+=1 means x = x+1, now the value of x = 1. After updating x , the condition is checked again. This process is repeated as long as the condition is true and once the condition becomes false the program breaks out of the loop . Of course, once a becomes equal to 5, we will no longer run through the loop.

## **while loop: break and continue:**

Python provides two keywords that **terminate** a loop iteration prematurely: break and continue.

* break leaves a loop.
* continue jumps to the next iteration.



Sometimes it's necessary to exit from a **Python while loop** before the loop has finished fully iterating over all the step values. This is typically achieved by a **"break"** statement.

**Example:**

**x=10**

**while True:**

**print (x)**

**x+=2;**

**if x>20:**

**break**

**print("After Break")**

**Output:**

10

12

14

16

18

20

After Break

In the above example, when the condition x>20, the break statement executed and immediately terminated the while loop and the program control resumes at the next statement.

### continue statement in Python while loop:

### continue statement in python while loop

The continue statement in **Python while loop** is used when we want to skip one or more statements in loop's body and to transfer the control to the next iteration.

**Example:**

x=0

while x < 50:

x+=10

if x==30:

continue

print (x)

print("Loop Over")

**Output:**

10

20

40

50

Loop Over