General Introduction

This course is an introduction to the Python programming language for CBSE Class 11 and 12. Python is an interpreted language. Interpreted languages do not need to be compiled to run. A program called an interpreter runs Python code on almost any kind of computer. This means that a programmer can change the code and quickly see the results. This also means Python is slower than a compiled language like C, because it is not running machine code directly.

Python is a good programming language for beginners. It is a high-level language, which means a programmer can focus on what to do, but does not require knowledge of computer hardware. Writing programs in Python takes less time than in some other languages.

Python is used by hundreds of thousands of programmers and is used in many places. Sometimes only Python code is used for a program, but most of the time it is used to do simple jobs while another programming language is used to do more complicated tasks.Its standard library is made up of many functions that come with Python when it is installed. On the Internet there are many other libraries available that make it possible for the Python language to do more things. These libraries make it a powerful language; it can do many different things.

Some things that Python is often used for are:

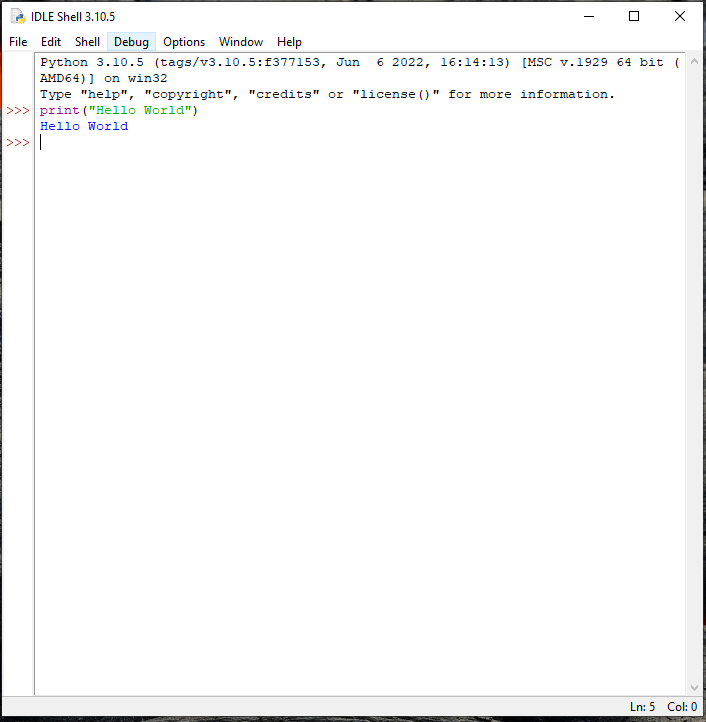
* Web development
* Scientific programming
* Data science
* Machine learning
* Numerical analysis
* Statistics
* Desktop GUIs applications
* Network programming
* Game programming.
* Complex algorithms creation
* Automation scripts
* Machine learning and artificial intelligence
* Audio and video applications

# Installation

Python can be downloaded from the official website <https://www.python.org/downloads/> . You can choose the installer for your current operating system. This will also install IDLE, the default Integrated Development Environment (IDE) for Python.

We will be using IDLE to run all our programs and exercise.

# Running your first Python script



# Execution mode

There are two ways to use the Python interpreter:

## Interactive mode

To work in the interactive mode, we can simply type a Python statement on the >>> prompt directly. As soon as we press enter, the interpreter executes the statement and displays the result(s)

## Script mode

In the script mode, we can write a Python program in a file, save it and then use the interpreter to execute it. Python scripts are saved as files where file name has extension “.py”.

Interactive mode allows execution of individual statement instantaneously. Whereas, Script mode allows us to write more than one instruction in a file called Python source code file that can be executed.

In what situation would you choose Script Mode over Interactive Mode?

# Keywords, identifiers and variables

Python **keywords** are special reserved words that have specific meanings and purposes and can’t be used for anything but those specific purposes.

As of Python 3.8, there are thirty-five keywords in Python. They are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [False](https://realpython.com/python-keywords/#the-true-and-false-keywords) | [await](https://realpython.com/python-keywords/#the-await-keyword) | [else](https://realpython.com/python-keywords/#the-else-keyword) | [import](https://realpython.com/python-keywords/#the-import-keyword) | [pass](https://realpython.com/python-keywords/#the-pass-keyword) |
| [None](https://realpython.com/python-keywords/#the-none-keyword) | [break](https://realpython.com/python-keywords/#the-break-keyword) | [except](https://realpython.com/python-keywords/#the-except-keyword) | [in](https://realpython.com/python-keywords/#the-in-keyword) | [raise](https://realpython.com/python-keywords/#the-raise-keyword) |
| [True](https://realpython.com/python-keywords/#the-true-and-false-keywords) | [class](https://realpython.com/python-keywords/#the-class-keyword) | [finally](https://realpython.com/python-keywords/#the-finally-keyword) | [is](https://realpython.com/python-keywords/#the-is-keyword) | [return](https://realpython.com/python-keywords/#the-return-keyword) |
| [and](https://realpython.com/python-keywords/#the-and-keyword) | [continue](https://realpython.com/python-keywords/#the-continue-keyword) | [for](https://realpython.com/python-keywords/#the-for-keyword) | [lambda](https://realpython.com/python-keywords/#the-lambda-keyword) | [try](https://realpython.com/python-keywords/#the-try-keyword) |
| [as](https://realpython.com/python-keywords/#the-as-keyword) | [def](https://realpython.com/python-keywords/#the-def-keyword) | [from](https://realpython.com/python-keywords/#the-from-keyword) | [nonlocal](https://realpython.com/python-keywords/#the-nonlocal-keyword) | [while](https://realpython.com/python-keywords/#the-while-keyword) |
| [assert](https://realpython.com/python-keywords/#the-assert-keyword) | [del](https://realpython.com/python-keywords/#the-del-keyword) | [global](https://realpython.com/python-keywords/#the-global-keyword) | [not](https://realpython.com/python-keywords/#the-not-keyword) | [with](https://realpython.com/python-keywords/#the-with-keyword) |
| [async](https://realpython.com/python-keywords/#the-async-keyword) | [elif](https://realpython.com/python-keywords/#the-elif-keyword) | [if](https://realpython.com/python-keywords/#the-if-keyword) | [or](https://realpython.com/python-keywords/#the-or-keyword) | [yield](https://realpython.com/python-keywords/#the-yield-keyword) |

**Identifiers** are names used to identify a variable, function, or other entities in a program.

The rules for naming an identifier in Python are as follows:

• The name should begin with an uppercase or a lowercase alphabet or an underscore sign (\_). This may be followed by any combination of characters a–z, A–Z, 0–9 or underscore (\_). Thus, an identifier cannot start with a digit.

• It can be of any length. (However, it is preferred to keep it short and meaningful).

• It should not be a keyword or reserved word.

• We cannot use special symbols like !, @, #, $, %, etc., in identifiers.

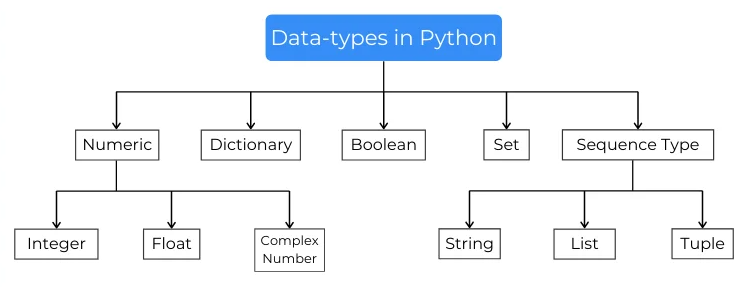
A variable in a program is uniquely identified by a name (identifier). Variable in Python refers to an item or element that is stored in the memory. Value of a variable can be a string (e.g., ‘b’, ‘NSCD’), numeric (e.g., 345) or any combination of alphanumeric characters (CD67). In Python we can use an assignment statement to create new variables and assign specific values to them.

You can declare and assign value to a variable with the assignment operator = :

a = 10

This will assign the value of 10 to the variable ‘a’

# Data Types



## Numeric Data

types are the data-type which represent the numeric values. It can be any number from the -infinity to +infinity. Numeric data-types are subdivided into three types:

### Integer

Integer data types are the one which can hold the integral value of the number.

a = 5

### Float

It is a real number with floating point representation. It is specified by a decimal point.

b = 5.0

### Complex Number

Complex number is represented by complex class. It is specified as (real part) + (imaginary part)j.

c = 2 + 4j

## Sequences

In Python, sequence is the ordered collection of similar or different data types. Sequences allows to store multiple values in an organized and efficient fashion. There are several sequence types in Python:

### String

A string is a collection of one or more characters put in a single quote, double-quote or triple quote.

### List

Lists are used to store values of same and different data types like integer, string, float, etc.

### Tuple

Just like list, tuple is also an ordered collection of Python objects. The only difference between tuple and list is that tuples are immutable i.e. tuples cannot be modified after it is created.

## Boolean

Boolean data-type is used to store values with True and False. True value is stored the 1 and False stores the 0. Boolean data-types are used to check conditions.

## Dictionary

Dictionaries are the unordered collection of data-types used to store the key-value pairs. Key in these dictionaries are the values that can be assigned to some data of any data-type, Whereas values are the information of the key.

## Set

Sets are Data-type like the list. In this type of data type, we can store different values which are non-repeating. Sets remove the duplicates of the elements and rearrange them in an unpredictable arrangement

# Input and Output

In Python, we have the input() function for taking the user input. The input() function prompts the user to enter data. It accepts all user input as string. The syntax for input() is:

input ([Prompt])

Python uses the print() function to output data to standard output device - the screen.

# Operators

### Arithmetic Operators

|  |  |  |
| --- | --- | --- |
| Operator | Description | Syntax |
| + | Addition: adds two operands | x + y |
| – | Subtraction: subtracts two operands | x – y |
| \* | Multiplication: multiplies two operands | x \* y |
| / | Division (float): divides the first operand by the second | x / y |
| // | Division (floor): divides the first operand by the second | x // y |
| % | Modulus: returns the remainder when the first operand is divided by the second | x % y |
| \*\* | Power: Returns first raised to power second | x \*\* y |

### Comparison Operator

|  |  |  |
| --- | --- | --- |
| Operator | Description | Syntax |
| > | Greater than: True if the left operand is greater than the right | x > y |
| < | Less than: True if the left operand is less than the right | x < y |
| == | Equal to: True if both operands are equal | x == y |
| != | Not equal to – True if operands are not equal | x != y |
| >= | Greater than or equal to True if the left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to True if the left operand is less than or equal to the right | x <= y |

### Assignment Operator

|  |  |  |
| --- | --- | --- |
| Operator | Description | Syntax |
| = | Assign value of right side of expression to left side operand | x = y + z |
| += | Add AND: Add right-side operand with left side operand and then assign to left operand | a+=b a=a+b |
| -= | Subtract AND: Subtract right operand from left operand and then assign to left operand | a-=b a=a-b |
| \*= | Multiply AND: Multiply right operand with left operand and then assign to left operand | a\*=b a=a\*b |
| /= | Divide AND: Divide left operand with right operand and then assign to left operand | a/=b a=a/b |
| %= | Modulus AND: Takes modulus using left and right operands and assign the result to left operand | a%=b a=a%b |
| //= | Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand | a//=b a=a//b |
| \*\*= | Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand | a\*\*=b a=a\*\*b |
| &= | Performs Bitwise AND on operands and assign value to left operand | a&=b a=a&b |
| |= | Performs Bitwise OR on operands and assign value to left operand | a|=b a=a|b |
| ^= | Performs Bitwise xOR on operands and assign value to left operand | a^=b a=a^b |
| >>= | Performs Bitwise right shift on operands and assign value to left operand | a>>=b a=a>>b |
| <<= | Performs Bitwise left shift on operands and assign value to left operand | a <<= b a= a << b |

### Logical Operators

|  |  |  |
| --- | --- | --- |
| Operator | Description | Syntax |
| and | Logical AND: True if both the operands are true | x and y |
| or | Logical OR: True if either of the operands is true | x or y |
| not | Logical NOT: True if the operand is false | not x |

### Identity Operators

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| is | True if the operands are identical (refer to the same object) | x is True |
| is not | True if the operands are not identical (do not refer to the same object) | x is not True |

### Membership Operators

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| in | True if value/variable is found in the sequence | 5 in x |
| not in | True if value/variable is not found in the sequence | 5 not in x |

### Precedence

Consider the following code:

result= 12+2-34\*23/4

Which mathematical operation will be evaluated first? To answer this question we have the order of precedence:

|  |  |
| --- | --- |
| Operator | Description |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Complement, unary plus and minus (method names for the last two are +@ and -@) |
| \* / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise 'AND' |
| ^ | | Bitwise exclusive `OR' and regular `OR' |
| <= < > >= | Comparison operators |
| <> == != | Equality operators |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

## Type Conversion

## Type Conversion is the conversion of object from one data type to another data type.

## There are two types of conversion:

## Implicit Conversion

## Explicit Conversion

## In Implicit type conversion, Python automatically converts one data type to another data type. Python avoids the loss of data in Implicit Type Conversion.

## Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.

## In Type Casting, loss of data may occur as we enforce the object to a specific data type.

## Errors

There are 3 types of errors in Python:

### Syntax

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

### Logical

A logical error is a bug in the program that causes it to behave incorrectly. A logical error produces an undesired output but without abrupt termination of the execution of the program.

### Runtime

Runtime error is when the statement is correct syntactically, but the interpreter cannot execute it. Runtime errors do not appear until after the program starts running or executing.

# Flow of Control

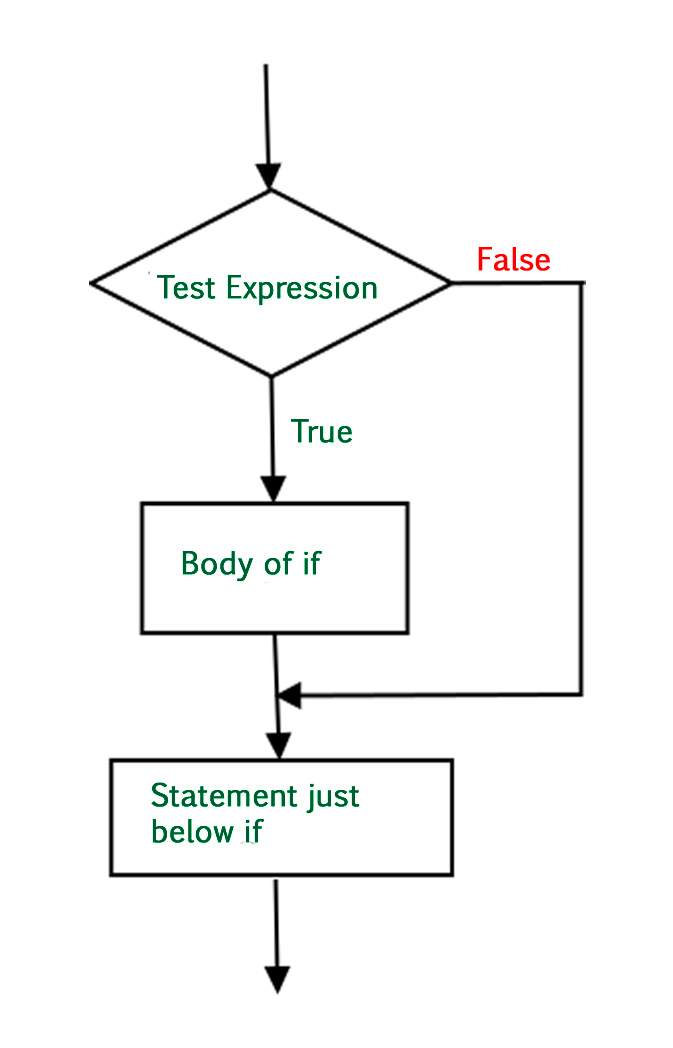
## if

It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise the statement will not be executed.

if condition:

statement1

statement2



## ****if-else****

We can use the else statement with if statement to execute a block of code when the condition is false.

if (condition):

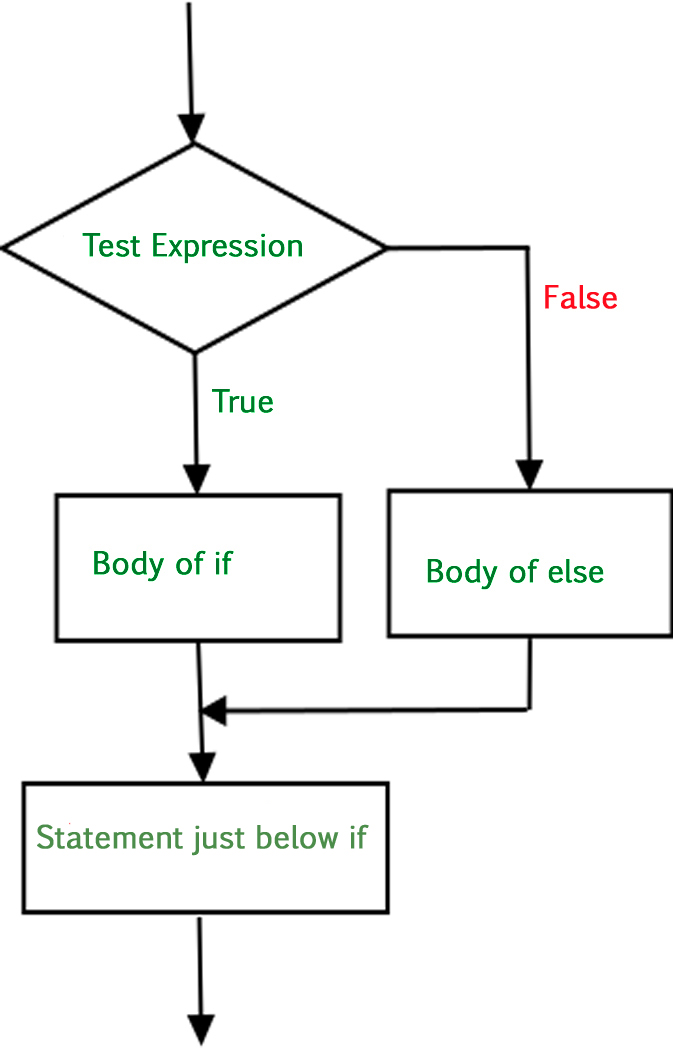
# Executes this block if

# condition is true

else:

# Executes this block if

# condition is false



## if-elif-else

Sometimes a situation arises when there are several conditions. To handle the situation Python allows adding any number of elif clause after an if and before an else clause. In the above case Python evaluates each expression (i.e. the condition) 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

if (condition):

statement

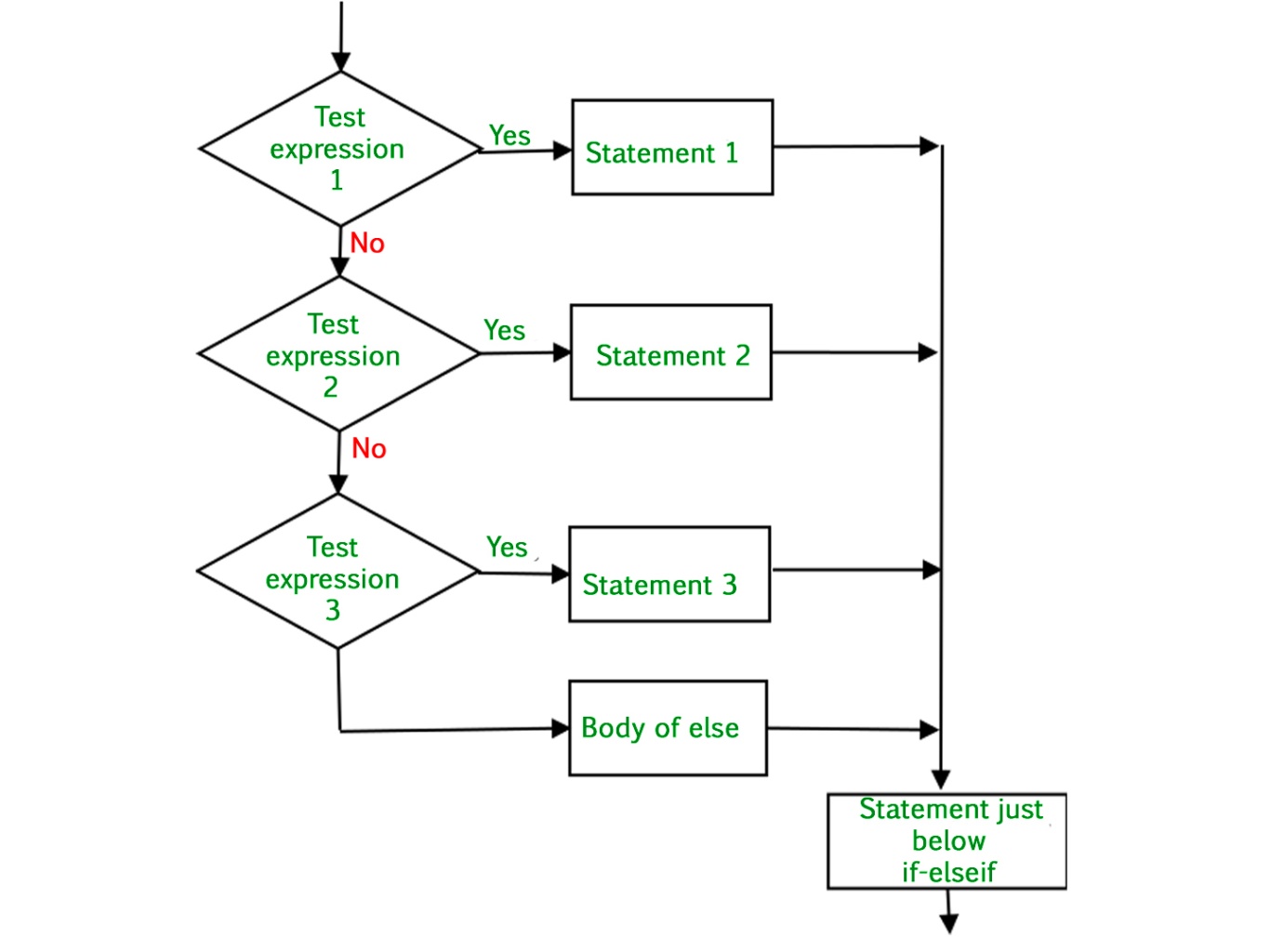
elif (condition):

statement

….

else:

statement



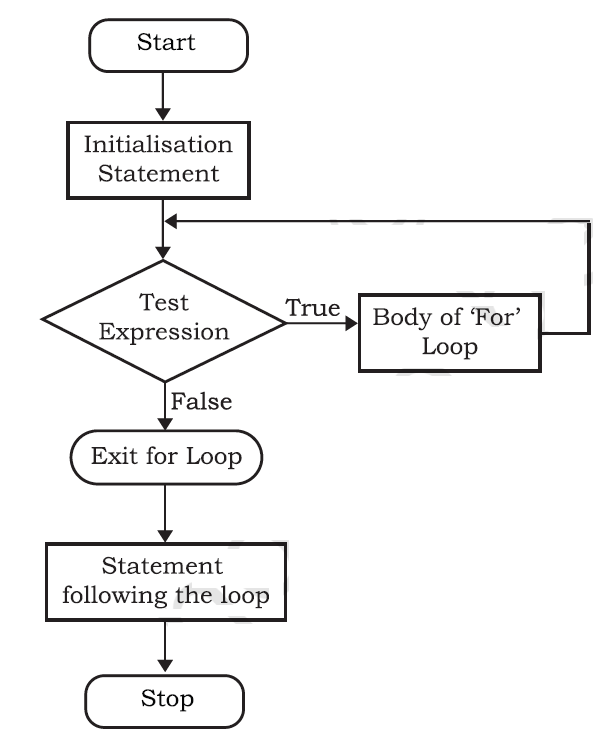
Note: Python can have nested if statement as well

# Loops

Python programming language provides the following types of loops to handle looping requirements. Python provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

## For

The for statement is used to iterate over a range of values or a sequence. The for loop is executed for each of the items in the range. These values can be either numeric, or, as we shall see in later chapters, they can be elements of a data type like a string, list, or tuple.



#### Two outputs on the same line:

By default, every **print()** statement will print in a new line. If you want two **print()** statements to print on the same line, you can use the end=’ ’. This will add a space between the two **print()** output instead of a new line. You can use any character or string after **end=**

for iterator\_var in sequence:

statements(s)

### The Range() Function

The range() is a built-in function in Python. Syntax of range() function is:

range([start], stop[, step])

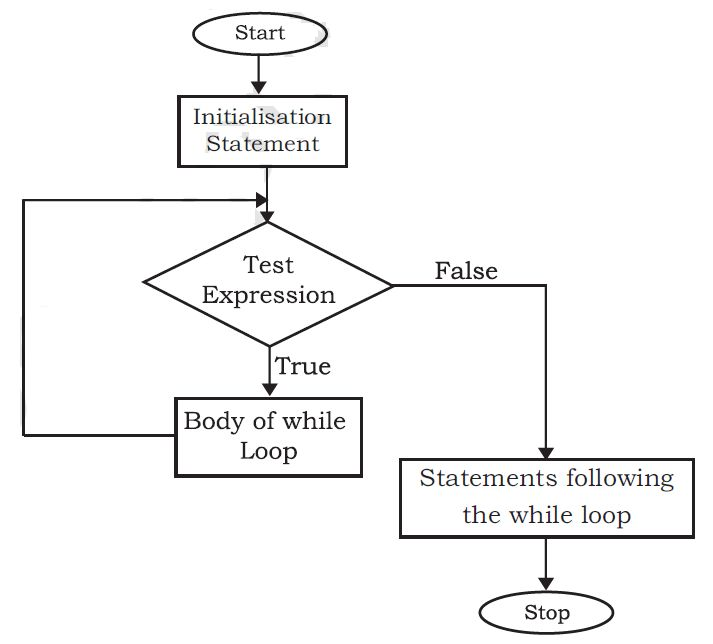
It is used to create a list containing a sequence of integers from the given start value upto stop value (excluding stop value), with a difference of the given step value.

list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## While

The while statement executes a block of code repeatedly as long as the control condition of the loop is true. The control condition of the while loop is executed before any statement inside the loop is executed.



In certain situations, when some particular condition occurs, we may want to exit from a loop (come out of the loop forever) or skip some statements of the loop before continuing further in the loop. These requirements can be achieved by using break and continue statements, respectively.

Break

The Break statement will terminate the current flow of execution and will exit from the loop

Continue

When a continue statement is encountered, the control skips the execution of remaining statements inside the body of the loop for the current iteration and jumps to the beginning of the loop for the next iteration.

Note: Similar to if statements, Python can have nested loops as well

# Function

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.

# We can define a function as follows

def fun():

print("Welcome to GFG")

# Then we can call it by using the function name

fun()

We can also pass arguments to a function

def printString(str):

print(str)

printStr(“Hello World”)

A function can also return a value

def numSqr(num):

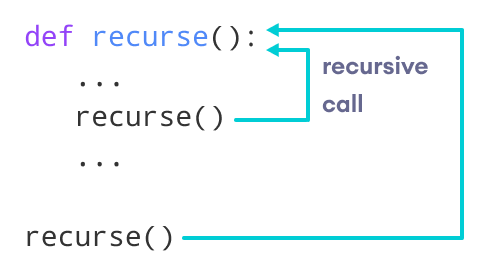
return num\*\*2

sq=numSqr(5)

print("Square is ",sq)

## Recursion

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly. Advantages of using recursion A complicated function can be split down into smaller sub-problems utilizing recursion. Sequence creation is simpler through recursion than utilizing any nested iteration. Recursive functions render the code look simple and effective.



# Strings

A string is a collection of one or more characters put in a single quote, double-quote or triple quote.

Syntax:

str1 = 'Hello World!'

# Lists

The data type list is an ordered sequence which is mutable and made up of one or more elements. Unlike a string which consists of only characters, a list can have elements of different data types, such as integer, float, string, tuple or even another list.

list1 = [2,4,6,8,10,12]

print(list1)

[2, 4, 6, 8, 10, 12]

# Tuple

A tuple is an ordered sequence of elements of different data types, such as integer, float, string, list or even a tuple. Elements of a tuple are enclosed in parenthesis (round brackets) and are separated by commas.

tuple1 = (1,2,3,4,5)

tuple1

(1, 2, 3, 4, 5)

|  |  |
| --- | --- |
| **Method** | **Description** |
| count() | Returns the number of times a specified value occurs in a tuple |
| index() | Searches the tuple for a specified value and returns the position of where it was found |

# Dictionary

Dictionaries are used to store data values in key:value pairs. A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.

dict3 = {'Mohan':95,'Ram':89,'Suhel':92,'Sangeeta':85}

dict3

{'Mohan': 95, 'Ram': 89, 'Suhel': 92, 'Sangeeta': 85}

# File Handling

There are mainly two types of data files

1. text file
2. binary file.

A text file consists of human readable characters, which can be opened by any text editor. On the other hand, binary files are made up of non-human readable characters and symbols, which require specific programs to access its contents. In Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)
3. Close the file

To open a file in Python, we use the open() function. The syntax of open() is as follows:

file\_object= open(file\_name, access\_mode)

Where the following access mode is supported:

|  |  |
| --- | --- |
| r | open an existing file for a read operation. |
| w | open an existing file for a write operation. If the file already contains some data then it will be overridden. |
| a | open an existing file for append operation. It won’t override existing data. |
| r+ | To read and write data into the file. The previous data in the file will be overridden. |
| w+ | To write and read data. It will override existing data. |
| a+ | To append and read data from the file. It won’t override existing data. |

Some example code for file handling are given below:

# Opening a file in read mode

file = open("file.txt", "r")

print (file.read())

# Python code to create a file (write mode)

file = open(‘newfile.txt’,’w’)

file.write("Writing content to a file")

file.close()

# Opening a file in append mode

file = open(append.txt','a')

file.write("This will be added to the existing file")

file.close()

The following are the methods available for a file

|  |  |
| --- | --- |
| **Method** | **Description** |
| close() | Closes the file |
| detach() | Returns the separated raw stream from the buffer |
| fileno() | Returns a number that represents the stream, from the operating system's perspective |
| flush() | Flushes the internal buffer |
| isatty() | Returns whether the file stream is interactive or not |
| read() | Returns the file content |
| readable() | Returns whether the file stream can be read or not |
| readline() | Returns one line from the file |
| readlines() | Returns a list of lines from the file |
| seek() | Change the file position |
| seekable() | Returns whether the file allows us to change the file position |
| tell() | Returns the current file position |
| truncate() | Resizes the file to a specified size |
| writable() | Returns whether the file can be written to or not |
| write() | Writes the specified string to the file |
| writelines() | Writes a list of strings to the file |

# Appendix

## String Operations

|  |  |
| --- | --- |
| **Function Name** | **Description** |
| capitalize() | Converts the first character of the string to a capital (uppercase) letter |
| casefold() | Implements caseless string matching |
| center() | Pad the string with the specified character. |
| count() | Returns the number of occurrences of a substring in the string. |
| encode() | Encodes strings with the specified encoded scheme |
| endswith() | Returns “True” if a string ends with the given suffix |
| expandtabs() | Specifies the amount of space to be substituted with the “\t” symbol in the string |
| find() | Returns the lowest index of the substring if it is found |
| format() | Formats the string for printing it to console |
| format\_map() | Formats specified values in a string using a dictionary |
| index() | Returns the position of the first occurrence of a substring in a string |
| isalnum() | Checks whether all the characters in a given string is alphanumeric or not |
| isalpha() | Returns “True” if all characters in the string are alphabets |
| isdecimal() | Returns true if all characters in a string are decimal |
| isdigit() | Returns “True” if all characters in the string are digits |
| isidentifier() | Check whether a string is a valid identifier or not |
| islower() | Checks if all characters in the string are lowercase |
| isnumeric() | Returns “True” if all characters in the string are numeric characters |
| isprintable() | Returns “True” if all characters in the string are printable or the string is empty |
| isspace() | Returns “True” if all characters in the string are whitespace characters |
| istitle() | Returns “True” if the string is a title cased string |
| isupper() | Checks if all characters in the string are uppercase |
| join() | Returns a concatenated String |
| ljust() | Left aligns the string according to the width specified |
| lower() | Converts all uppercase characters in a string into lowercase |
| lstrip() | Returns the string with leading characters removed |
| maketrans() | Returns a translation table |
| partition() | Splits the string at the first occurrence of the separator |
| replace() | Replaces all occurrences of a substring with another substring |
| rfind() | Returns the highest index of the substring |
| rindex() | Returns the highest index of the substring inside the string |
| rjust() | Right aligns the string according to the width specified |
| rpartition() | Split the given string into three parts |
| rsplit() | Split the string from the right by the specified separator |
| rstrip() | Removes trailing characters |
| splitlines() | Split the lines at line boundaries |
| startswith() | Returns “True” if a string starts with the given prefix |
| strip() | Returns the string with both leading and trailing characters |
| swapcase() | Converts all uppercase characters to lowercase and vice versa |
| title() | Convert string to title case |
| translate() | Modify string according to given translation mappings |
| upper() | Converts all lowercase characters in a string into uppercase |
| zfill() | Returns a copy of the string with ‘0’ characters padded to the left side of the string |

## List Operations

|  |  |
| --- | --- |
| **Method** | **Description** |
| append() | Adds an element at the end of the list |
| clear() | Removes all the elements from the list |
| copy() | Returns a copy of the list |
| count() | Returns the number of elements with the specified value |
| extend() | Add the elements of a list (or any iterable), to the end of the current list |
| index() | Returns the index of the first element with the specified value |
| insert() | Adds an element at the specified position |
| pop() | Removes the element at the specified position |
| remove() | Removes the item with the specified value |
| reverse() | Reverses the order of the list |
| sort() | Sorts the list |

## Dictionary Operations

|  |  |
| --- | --- |
| **Method** | **Description** |
| clear() | Removes all the elements from the dictionary |
| copy() | Returns a copy of the dictionary |
| fromkeys() | Returns a dictionary with the specified keys and value |
| get() | Returns the value of the specified key |
| items() | Returns a list containing a tuple for each key value pair |
| keys() | Returns a list containing the dictionary's keys |
| pop() | Removes the element with the specified key |
| popitem() | Removes the last inserted key-value pair |
| setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| update() | Updates the dictionary with the specified key-value pairs |
| values() | Returns a list of all the values in the dictionary |