# UNIT-1

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

**Syllabus: Introduction to Python Programming**: Python Interpreter, Parts of Python Language: Identifiers, Keywords, Statements, Variables and Operators, Data Types, Indentation, Comments, Type conversions, type() Function and is Operator, Dynamic and strong Typed language, Command line arguments.
**Control Flow Statements**: if, if...else, if...elif...else Decision Control Statements, Nested if Statement, the while Loop, the for Loop, the continue and break Statements. Exception Handling.

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

## Introduction

Python is a general-purpose programming language with simple syntax. It has efficient high-level data structures and supports object-oriented programming. It is freely available for platforms like Windows, Linux and Mac OS. Standard python language comes with a rich set of built-in libraries. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

## History

Python was created by **Guido van Rossum.** Python was conceived in the late 1980's and its implementation was started in December 1989 at CWI in the Netherlands as a successor to the ABC programming language.

- The first ever version of Python (i.e., Python 1.0) was introduced in 1991.
- Python 2.0 was released on 16 October 2000.
- Python 3.0 was released on 03 December 2008.
- The evolution of Python has reached up to Version 3.x (till 2018). Latest version of python is 3.13.7.

## Python Interpreter

Interpreter is a computer programs that converts the source code(high level language) into machine level language and executes line by line.

Like compilers, interpreters are also called as translators. Executing statements line by line is called as interpretation. Programming languages like Python, Ruby, PHP and JavaScript uses interpreters.

Python interpreter (CPython) is written in C programming language. It reads the source code line by line and checks indentation rules and syntax errors. if there is any error, Python stops its execution and displays the error information.

When a Python Program/script is executed, the interpreter goes through the below steps:

- **Source Code Analysis:** The interpreter reads the source code and converts it into some sort of token. This process is Lexical Analysis.

- **Parsing**: Tokens are parsed into a structure which is called the Abstract Syntax Tree or AST to identify syntactical mistakes.
- **Compilation**: Statements are converted into a low-level code known as bytecode.
- **Execution**: The bytecode is executed by the Python Virtual Machine (PVM), which translates the bytecode into machine code and runs it.

This process is repeated for anytime execution of Python program/script. This makes the programming dynamic.

## Parts of Python Language

## Identifiers

An identifier is a name given to any user-defined variable/function/class/module. Identifiers may be one or more characters.

Rules to be followed in writing an identifier in python:

- Identifier can be a combination of letters (lower and uppercase) and digits, underscore (_).
- Identifier must begin with an alphabet or an underscore.
- Cannot start with a digit however digit can be allowed elsewhere.
- **Keywords/reserve** words cannot be used as identifiers.
- **Whitespaces** and special symbols like **!, @,$,#,%** are not allowed in identifiers.
- No restriction on length of an identifier.

Valid identifier examples: **myCountry, other_1 MyCountry and good_morning**.

Invalid identifier examples: **my country, 3names, !abc, $name, @a, #num, %h…….**

## Keywords

Keywords are a list of reserved words that have predefined meaning in python.

- Keywords are special vocabulary and cannot be used as variable names, function names, constants, class names and module names. Attempting to use a keyword as an identifier name will cause an error.

List of Keywords:

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

## Statements

A statement in python program is an instruction given to a python interpreter to accomplish a task.

Example:

```
>>>z = 1
```

```
>>>y = 2
>>>x = y + z
```

Python program is a collection of statements. Statements are everything that can make up a line (or several lines) of Python code.

Expressions are statements as well. Expression is an arrangement of values and operators which are evaluated to make a new value.

## Variables

In python a Variable is an user defined identifier/placeholder which is used to hold any type of data. In program variables are used to assign and modify data during execution.

In Python, there is no need to declare a variable explicitly by specifying the type of data it hols. To define a new variable in Python, we simply assign a value to a name. Based on the data it holds, type will be automatically decided by python interpreter. To define/declare a new variable in Python, simply assign a value to a name/identifier.

Follow the below rules for creating legal variable names in Python:

- Variable names can consist of any number of letters, underscores and digits.
- Variable should not start with a number.
- Python Keywords are not allowed as variable names.
- Variable names are case-sensitive. For example, computer and Computer are different
- variables.

Best practices while naming a variable(not compulsory):

- Python variables use lowercase letters with words separated by underscores as necessary to improve readability.
- Avoid naming a variable where the first character is an underscore.
- Ensure variable names are descriptive and clear enough. This allows other programmers
- to have an idea about what the variable is representing.

**Assigning Values to Variables:**
The general format for assigning values to variables is as follows:

***Syntax: variable_name = expression/value***

The equal sign '=' also known as assignment operator is used to assign values to variables. The operand to the left of the '=' operator is the name of the variable and the operand to the right of the '=' operator is the expression which can be a value or any code snippet that results in a value.

## Operators

Operators are symbols that perform certain mathematical or logical operation to manipulate data values and produce a result based on some rules.

Python language supports a wide range of operators. They are

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators (in, not in)
7. Identity Operators (is, is not)
8. ternary operator (true_value ***if*** condition ***else*** false_value)

**1. Arithmetic Operators**

Arithmetic operators are used to execute arithmetic operations such as addition, subtraction, division, multiplication etc.

List of Arithmetic Operators

If the value of x is 5 and y is 3.

| Operator | Name of the operator | Description | Example |
|---|---|---|---|
| + | Addition operator | Adds two operands, producing their sum. | x+y=8 |
| - | Subtraction operator | Subtracts the two operands, producing their difference | x-y=2 |
| * | Multiplication operator | Produces the product of the operands | x*y=15 |
| / | Division operator | Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor | x/y=1.66 |
| % | Modulus operator | Divides left hand operand by right hand operand and returns a remainder | x%y=2 |
| ** | Exponent operator | Performs exponential (power) calculation on operators | x**y=125 |
| // | Floor division operator | Returns an integral part of the quotient | x//y=1<br>5.0//3.0=1.0 |

**2. Assignment Operators**

Assignment operators are used for assigning the values generated after evaluating the right operand to the left operand. Assignment operation always works from right to left. Assignment operators are either simple assignment operator or compound assignment operators.

- Simple Assignment operator: x=1, x=x+1
- Compound Assignment operator: x += 1

List of Assignment operators:

| Operator | Name of the operator | Description | Example |
|---|---|---|---|
| = | Assignment | Assigns values from right side operands to left side operand. | z = p + q |
| += | Addition Assignment | Adds the value of right operand to the left operand and assigns the result to left operand. | z += p<br>(z = z + p) |

| | | | |
|---|---|---|---|
| -= | Subtraction Assignment | Subtracts the value of right operand from the left operand and assigns the result to left operand. | z -= p (z = z - p) |
| *= | Multiplication Assignment | Multiplies the value of right operand with the left operand and assigns the result to left operand. | z *= p (z = z * p) |
| /= | Division Assignment | Divides the value of right operand with the left operand and assigns the result to left operand. | z /= p (z = z / p) |
| **= | Exponentiation Assignment | Evaluates to the result of raising the first operand to the power of the second operand. | z **= p (z = z ** p) |
| //= | Floor Division Assignment | Produces an integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor. | z //= p (z = z // p) |
| %= | Remainder Assignment | Computes the remainder after division and assigns the value to the left operand. | z %= p (z = z % p) |

**3. Comparison Operators**

Comparison operators are used to compare the values of two operands in an expression. The output of these comparison operators is always a Boolean value, either True or False. The operands can be Numbers or Strings or Boolean values. Strings are compared letter by letter using their ASCII values.

List of Comparison Operators: If the value of p is 10 and q is 20.

| Operator | Name of the operator | Description | Example |
|---|---|---|---|
| == | Equal to | If the values of two operands are equal, then the condition becomes True. | p==q =>False |
| != | Not Equal to | If values of two operands are not equal, then the condition becomes True. | p!=q=>True |
| < | Less than | If the value of left operand is greater than the value of right operand, then condition becomes True. | p<q => True |
| > | Greater than | If the value of left operand is less than the value of right operand, then condition becomes True. | p>q=>False |
| <= | Less than or equal to | If the value of left operand is greater than or equal to the value of right operand, then condition becomes True. | p<=q=>True |
| >= | Greater than or equal | If the value of left operand is less than or equal to the value of right operand, then condition becomes True. | p>=q=>False |

**4. Logical Operators**

The logical operators are used for comparing/negating the logical values of their operands and returns the resulting logical value. The values of the operands must evaluate to either True or False.

The result of the logical operator is always a Boolean value, True or False.

List of Logical Operators: if the boolean value of p is True and q is False.

| **Operator** | **Name of the operator** | **Description** | **Example** |
|---|---|---|---|
| and | Logical AND | Performs AND operation and the result is True when both operands are True | p and q => False |
| or | Logical OR | Performs OR operation and the result is True when any one of the operands is True | p or q => True |
| not | Logical NOT | Reverses the operand state | not p => False |

Truth table for Boolean Logic

| **P** | **Q** | **P and Q** | **P or Q** | **not P** |
|---|---|---|---|---|
| True | True | True | True | False |
| True | False | False | True | |
| False | True | False | True | True |
| False | False | False | False | |

Logical expressions are evaluated from left to right.

They are used in "short-circuit valuation" using the following rules:

1. False and (some_expression) in short-circuit always evaluated to False.
2. True or (some_expression) in short-circuit always evaluated to True.

**Bitwise Operators**

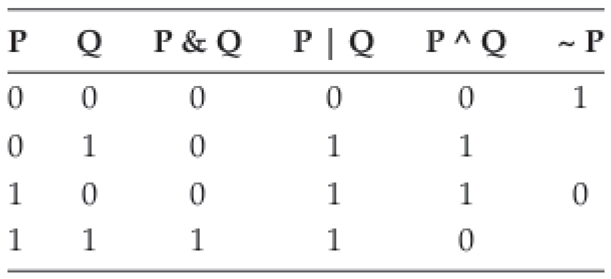
Bitwise operators treat their operands as a sequence of bits (zeroes and ones) and perform bit by bit operation.

List of Bitwise Operators:

Let p is 60 and q is 13.

| **Operator** | **Name of the Operator** | **Description** | **Example** |
|---|---|---|---|
| & | Bitwise AND | Returns 1 if both the bits are 1 | p & q =12 |
| \| | Bitwise OR | Returns 1 if one of the bits is 1 | p \| q = 61 |
| ^ | XOR | Returns 1 if one of the bits is 1 otherwise 0 | p ^ q = 49 |
| ~ | Bitwise Not | Inverts the bits | ~p = -61 in Complement form |
| << | Left Shift | Shift the bits towards left side by specified number and add's 0 on the right side | p<<2 = 240 |
| >> | Right Shift | Shift the bits towards right side by specified number and add's 0 on the left side | p>>2 = 15 |

Bitwise Truth Table



| P | Q | P & Q | P \| Q | P ^ Q | ~ P |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | |

**Membership Operator**

These operators are used to check whether the left side operand is part of the right side operand or not.

- **in** and **not in** are called as membership operators in python.

**Example:**

```
x = "Welcome to Python Class"
y = "to"
if x in y:
        print(" y is available in x")
else:
        print("y is not available in x")
```

Output: y is available in x

**Identity Operator**

These operators are used to compare the memory locations of the two operators to check that they are identical or not.

- **is** and **is not** are called as identity operators in python

**Example:**

```
a= b= 10
if (a is b):
   print("Identical")
else:
  print("Not Identical")
```

Output: Identical

**Example:**

```
a = 10
c = 20
if c is not a:
   print("Not Identical")
else:
```

```
                print("Identical")
```
Output: Not Identical

**Ternary Operator**

In Python, the ternary operator is a concise way to perform conditional checks and assign values or execute expressions based on a condition. It is also known as a conditional expression.

Syntax:

```
variable = true-value if condition else false_value
```

**Example:**

```
a = 30
b = 40
max = a if a > b else b
print("Max is :", max)
min = a if a < b else b
print("Min is :", min)
```
Output:
```
Max is : 40
Min is : 30
```

## Data Types

Data types specify the type of data that can be stored and manipulated within a python program.

Since everything is an object in Python programming, data types are actually classes and variables are instances(object) of these classes.

Basic data types of python are:

1. Numbers
2. Strings
3. Boolean
4. None

**1. Numbers:** Numbers type data includes Integers, floating point numbers and complex numbers. They are called as int, float and complex classes in Python.

- No restriction on the length of Integers.
- A floating point number is accurate up to 15 decimal places. Integer part and floating part are separated by decimal point.
- Complex numbers are written in the form of x + yj, where x is the real part and y is the imaginary part.

```
>>> a =10
>>> type(a)
    <class 'int'>
>>> b = 20.5
>>> type(a)
    <class 'float'>
>>> c = complex(3,5)
>>> type(a)
    <class 'complex'>
```

**2. Strings:** A string is a character or a sequence of characters which includes letters, numbers, whitespaces and other types.

- They are also called as string literal.
- Single quotes or double quotes are used to represent strings.
- Multiline strings can be denoted using triple quotes ''' or """.

```
>>> str1 = 'Hello World'
>>> str2 = "Hello, how are you?"
>>> type(str2)
    <class 'str'>
```

**3. Booleans:** A Boolean variable can only have two possible values, as represented by the keywords, True and False. Python bool() function allows an expression to be evaluated to either True or False.

```
>>> a = True
>>> b = False
>>> type(a)
    <class 'bool'>
```

**4. None:** None is a special data type in python. It is used to represent the absence of a value.

```
>>> a = None
>>> type(a)
    <class 'NoneType'>
```

## Indentation

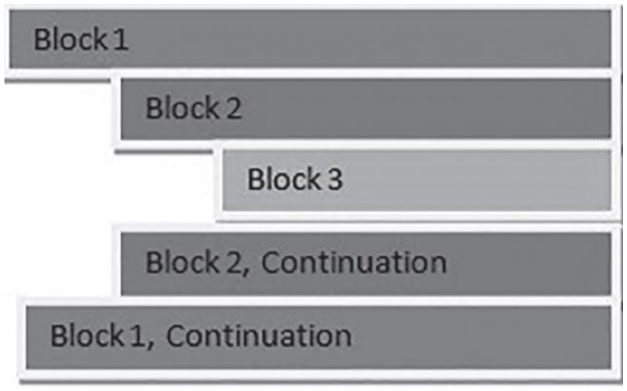
In Python, indentation is a fundamental part of the language's syntax. Unlike many other programming languages where indentation is optional and used solely to improve code clarity and readability.

Python uses indentation to define blocks of code. Indentation is required to indicate blocks of code for constructs like if, for, while, and functions.

If the indentation is inconsistent or missing, Python will raise a Syntax Error.

To generate indentation ':' (colon) symbol is used in python.

1 tab space is used as one indentation.



In the above diagram, Block 2 and Block 3 are nested under Block 1. Usually, four whitespaces are used for indentation.

## Comments

Comment lines are used to provide the description of a statement.

Single line comments:

‘#’ is used to create single line comments.

Multi line comments:

Pair of three single quotes or pair of three double quotes are used to create muti line comments in python.

Example:

```
''' multi
  line
  comments'''
```

```
""" multi
  line
  comments """
```

## Type conversions

Changing the datatype of a variable into another type is called as type conversion. Python supports two types of conversions.

1. Implicit conversion
2. Explicit Conversion (Type Casting)

For compatible types, interpreter automatically converts lower types of data into higher types. This kind of conversion is called implicit conversion.

Example:

```
>>> a = 10
>>> b = 20.34
>>> c = a + b
>>> c
    30.34
```

For incompatible types of operands, before performing any operation on operands type is to changed. To change the type of an operand, python provides the following type conversion functions:

1. int(): It is used to convert float and string type of data into integer type
2. float(): It is used to covert int and string type of data into floating type
3. str(): It is used to convert any type of data into string type.
4. chr(): It is used to convert an integer value (range: 0 -255) into its equivalent ASCII value
5. ord(): It is used to convert an integer value into its equivalent Unicode value.
6. oct(): It is used to convert decimal value into octal value. ‘0o’ is prefixed with the result
7. hex(): It is used to convert decimal value into hexadecimal value. ‘0x’ is prefixed with the result.
8. complex(real, img): It is used to create complex data of the form ‘real+img*j’. First argument can be string or integer. Second argument cannot be a string.

## type() Function

To know the datatype of a variable or what kind of data is stored in a particular variable, type() is used.

The syntax for type() function is:

```
type(object/variable)
```

The type() function returns the data type of the given object/variable.

```
>>> type(1)
        <class 'int'>
>>> type(6.4)
        <class 'float'>
>>> type("A")
        <class 'str'>
>>> type(True)
        <class 'bool'>
```

## ‘is’ Operator

‘is’ and ‘is not’ operators are called as identity operators in python. They are used to compare the memory locations of the operands.

- 'is' operator returns True if the values of operands on either side of the operator point to the same object and False otherwise.
- 'is not' operator evaluates to False if the values of operands on either side of the operator point to the same object and True otherwise.

## Dynamic and strong Typed language

Python is a dynamic language as the type of the variable is determined during run-time. Python is also a strongly typed language as the interpreter keeps track of all the variables types. In strongly typed language, operations are not allowed on incompatible type of data.

**Example:**

If we try to add an integer and a string as below, python interpreter throws an error.

```
>>> 1 + "a"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

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

## Command line arguments

Arguments which are passed to python interpreter at the time of running the program are called as command line arguments. They are stored in a predefined list called ‘**argv**’ which is available in ‘**sys**’ module.

```
argv = [‘program_name.py’, arg1, arg2,........, argn]
```

First argument (sys.argv[0]) is always the name of the program and the subsequent elements are the arguments passed to the python program.

Command line arguments are passed to a program use the below command at the time of running the python program:

```
python program_name.py arg1 arg2 arg3 . . . . argn
```

To print all the command line arguments passed to a program use the below statement.

```
print(“command line arguments:”, sys.argv)
```

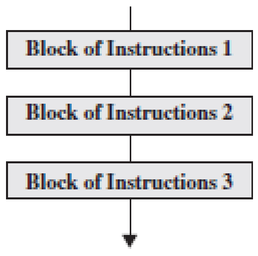
**Program to demonstrate command line arguments**

```
#Area of Triangle
import sys
b = int(sys.argv[1])
h = int(sys.argv[2])
area = 0.5*b*h
print(“Area of a triangle is :”, area)
```

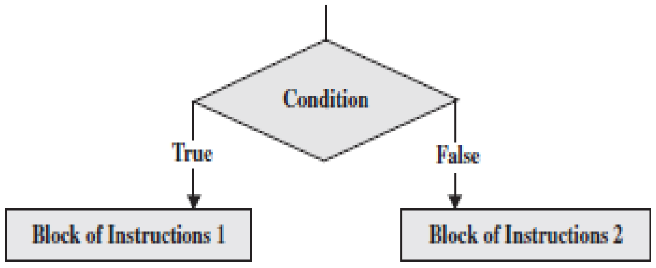
## Control Statements

Python program is a collection of various kinds of statements. In general, most of the statements are executed sequentially from top to bottom in the order as written in the program. Along with the sequential statements, programs often contain statements that controls the flow of executions. They are:

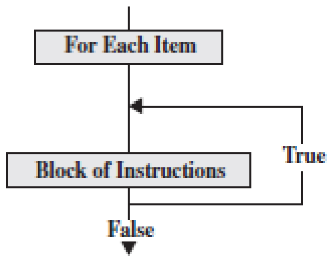
1. Decision Making Statements/Conditional Statements
2. Looping Statements/Iterative Statements
3. Loop Control Statements



Sequential



Decision-Making



Looping

1. **Decision Making Statements**: Depending on the result (True/False) of condition, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if...else and if...elif...else).
2. Looping Statements: This control structure allows the execution of a block of statements multiple times until a loop termination condition is met (for loop and while loop). Loop Control Flow Statements are also called Repetition statements or Iteration statements.
3. Loop Control Statements: These statements are essential programming constructs that allow the developer to control the flow of iterations in loops. In Python, there are three primary loop control statements ('break', 'continue', 'pass').

Decision Making Statements

These statements are used to make decisions based on certain conditions.

The following are the conditional statements available in Python:

- if
- if...else
- if.....elif.....else
- Nested if

**The *if* Statement**

The if statement executes a block of code if a condition is True.

The syntax for if statement is,

***if Boolean_ Expression:***

***statement (s)***

These statement starts with if keyword and ends with a colon (:). The expression in an if statement should be a Boolean expression. If the Boolean expression evaluates to True then statements in the if block will be executed.

For example,

```
>>> if 20 > 10:
... print(f"20 is greater than 10")
20 is greater than 10
```

```
#Program to read a Number and Prints a Message If It Is Positive
number = int(input("Enter a number"))
if number >= 0:
    print(f"The number entered by the user is a positive number")
```

**The if...else Decision Control Statement**

An *if* statement can also be followed by an *else* statement which is optional. An else statement does not have any condition. Statements in the *if* block are executed if the Boolean_Expression is True. If the Boolean_Expression is False, statements in the *else* block will be executed. if...else statement allows two-way decision making.

The syntax for if...else statement is,

```
if Boolean_Expression:
    statement'(s)
else:
    statement'(s)
```

For Example:

```
#Program to Find If a Given Number Is Odd or Even
number = int(input("Enter a number"))
if number % 2 == 0:
    print(f"{number} is Even number")
else:
    print(f"{number} is Odd number")
```

**The if...elif...else Decision Control Statement**

The *if...elif...else* is also called as multi-way decision control statement. To choose one from several possible alternatives, then an *elif* statement is used along with an *if* statement. The keyword ‘elif’ is short form of ‘else if’.

The syntax for *if...elif...else* statement is,

```
if Boolean_Expression_1:
        statement_1
elif Boolean_Expression_2:
        statement_2
elif Boolean_Expression_3:
        statement_3
:
else:
        statement_last
```

if there are multiple Boolean expressions in a decision control statement, only the logical Boolean expression which is evaluated to True will be executed.

- If Boolean_Expression_1 is True, then statement_1 is executed.
- If Boolean_Expression_1 is False and Boolean_Expression_2 is True, then statement_2 is executed.
- If Boolean_Expression_1 and Boolean_Expression_2 are False and Boolean_
- Expression_3 is True, then statement_3 is executed and so on.
- If none of the Boolean_Expression is True, then statement_last is executed.

Example:

#Program to Prompt for a Score between 0 and 100. If the Score is out of Range, Print an Error. If the Score Is between 0 and 100, Print the Grade as (>=90 as A+, >=80 as A, >=70 as B, >=60 as C, >=50 as D, >=40 as E, for other number as F).

```
marks = float(input("Enter your score"))
if marks < 0 or marks > 100:
        print('Marks must be in the range of 0 to 100')
elif marks >= 90:
        print('Your Grade is "A" ')
elif marks >= 80:
        print('Your Grade is "B" ')
elif marks >= 70:
        print('Your Grade is "C" ')
elif marks >= 60:
        print('Your Grade is "D" ')
elif marks >= 50:
        print('Your Grade is "E" ')
else:
        print('Your Grade is "F" ')
```

**Nested *if* Statement**

Sometimes, it is required to place an *if* statement inside another *if* statement. An if statement that contains another if statement either is called a Nested if statement. It is also called as nesting of *if* statements.

The syntax of the nested if statement is,

```
if Boolean_Expression_1:
    if Boolean_Expression_2:
        statement_1
    else:
        statement_2
else:
    statement_3
```

If the Boolean_Expression_1 is True, then the control shifts to Boolean_Expression_2 and if the expression is evaluated to True, then statement_1 is executed, if the Boolean_Expression_2 is evaluated to False then the statement_2 is executed. If the Boolean_Expression_1 is evaluated to False, then statement_3 is executed.

Example:

#Program to Check Whether the Given Year Is a Leap Year or not

```
year = int(input('Enter a year'))
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print(f'{year} is a Leap Year')
        else:
            print(f'{year} is not a Leap Year')
    else:
        print(f'{year} is a Leap Year')
else:
    print(f'{year} is not a Leap Year')
```

## Looping Statements

Sometimes we may need execute a statement or a set of statements several numbers of times. To do this we can go for loop statements.

Python provide various types of loops which can repeat some specific code several numbers of times. They are:

- while loop
- for loop

**While loop**

Using while loop we can execute a set of statements if the condition is true. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax for while loop is,

```
while Boolean_Expression:
statement(s)
```

**Example:**

```
i = 0
while i < 10:
    print(f"Current value of i is {i}")
    i = i + 1
```

If the Boolean expression evaluates to True, then the while loop block is executed. After each iteration of the block, the Boolean expression is again checked, and if it is True, the loop is iterated again.

Each repetition of the loop block is called an iteration of the loop.

**#Write a Program to Find the Average of n Natural Numbers where n Is the Input from the User**

```
number = int(input("Enter a number up to which you want to find the
average"))
i = 0
sum = 0
count = 0
while i < number:
    i = i + 1
    sum = sum + i
    count = count + 1
average = sum/count
print(f"The average of {number} natural numbers is {average}")
```

**#Program to Find the GCD of Two Positive Numbers**

```
m = int(input("Enter first positive number"))
n = int(input("Enter second positive number"))
if m == 0 and n == 0:
    print("Invalid Input")
if m == 0:
    print(f"GCD is {n}")
if n == 0:
    print(f"GCD is {m}")
```

```
while m != n:
        if m > n:
                m = m – n
        if n > m:
                n = n – m
print(f"GCD of two numbers is {m}")
```

**For loop**

For loop is used to repeat a statement of set of statements for a specific number of times.

The syntax for the *for* loop is,

```
for iteration_variable in sequence:

statement(s)
```

The for statement starts with for keyword and ends with a colon. iteration_variable can be any valid user defined variable name. The first item in the sequence is assigned to the iteration_variable. so, the block of for loop will be executed for each element in the sequence.

Sequence can be a string literal or any object that holds a sequence of elements.

**range() function:**

It's a built-in function used in for loop. it generates a sequence of integers based on the arguments passed to it.

The syntax for range() function is,

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

Both start and step arguments are optional and the arguments value should always be an integer.

- start value indicates the beginning of the sequence. If the start argument is not specified, then the sequence of numbers start from zero by default.
- stop value indicates the value to which sequence is to be generated but not including the number itself.
- step value indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

In the above syntax, start and step are optional arguments, they can be ignored also.

- **range(stop):** generates sequence of integers from 0 to stop-1
- **range(start, stop):** generates sequence of integers from start to stop-1

- **range(start, stop, step)**: generates sequence of integers from start to stop-1 with step as increment.

Example:

#Program to Iterate through Each Character in the String Using for Loop

```
for i in "Hyderabad":
    print(i)
```

'''Program to Find the Sum of All Odd and Even Numbers up to a Number Specified by the User.'''

```
number = int(input("Enter a number"))
even = 0
odd = 0
for i in range(number+1):
    if i % 2 == 0:
        even = even + i
    else:
        odd = odd + i
print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")
```

**Note:** All the looping statements can have an optional block called '***else***' block. Code in the ***else*** block will be executed only if the loop finishes successfully. If the loop is terminated by a break statement, the else block is skipped off.

## Loop Control Statements

The ***break*** and ***continue*** statements provide greater control over the execution of code in a loop. So, they are called as loop control statements.

Note: Break and Continue must be used inside while and for statements only.

**Break Statement:**

Whenever the break statement is encountered, the control immediately stops the execution of the loop and comes out of the loop.

Syntax:

```
if condition:
    break
```

Example:

```
        n = 0
```

```
while True:
    print(f"The latest value of n is {n}")
    n = n + 1
    if n > 5:
        print(f"The value of n is greater than 5")
        break
print("While loop is ended")
```

In the above sample code, when the 'n' value becomes 6, break statement is executed, control comes out of the loop.

**Continue statement:** To pass control to the next iteration without exiting the loop, continue statement is used.

Continue statement is used to stop/skip the current iteration and move the control to first statement of the loop to start next iteration.

Syntax:

```
if condition:
    continue
```

Example:

```
for i in range(1,10):
    if i == 5:
        continue
    print(i)
```

**#program to check whether the given integer is prime or not**

```
n = int(input("enter n value"))
prime = True
for i in range(2, n):
    if n % i == 0:
        prime = False:
        break
if prime:
    print(f"{n} is a prime number")
else:
    print(f"{n} is not a prime number")
```

**Pass Statement:**

The pass statement in Python is a placeholder that does nothing when executed. This statement is used to write empty block in the program without getting any syntactical errors.

Commonly used in Conditional statements, loops, functions, and classes, loops, as a placeholder.

Syntax:

***pass***

Example:

1) if True:
```
    pass
```
2) for i in range(5):
```
    pass
```
3) def func_name():
```
    pass
```
4) class class_name:
```
    pass
```

So, the pass statement is a simple yet powerful tool for writing clean and organized code during software development.

## Match Statement in Python

The match statement, introduced in Python 3.10, is a powerful feature for pattern matching, allowing you to write cleaner and more expressive conditional logic. It works similarly to a switch statement in other languages. It is also used to select one option out of n available options.

Syntax:

```
match expression:
  case pattern1:
    # Code block for pattern1
  case pattern2:
    # Code block for pattern2
  .
  .
  case _:
    # Default case (optional)
```

Example:

```
#program to check whether the given integer is positive/negative/zero
num=int(input("Enter a integer value"))
match num:
    case n if n > 0:
      print("Positive number")
    case n if n < 0:
      prnt("Negative number")
    case _:
      print("Zero")
```

Example:

```
n = int(input("enter a number between 1 to 7: "))
match n:
  case 1:
    print("Week day is Sunday")
  case 2:
    print("Week day is Monday")
  case 3:
    print("Week day is Tuesday")
  case 4:
    print("Week day is Wednesday")
  case 5:
    print("Week day is Thursday")
  case 6:
    print("Week day is Friday")
  case 7:
    print("Week day is Saturday")
  case _:
    print(" Only enter number between 1 and 7")
```

## Exception Handling

**Introduction:**

Errors in python are categorized into two.

- Syntax Errors: Errors due mistake done while writing the syntax of a statement
- Runtime Errors: Errors that appear during the running time of the program

Exception is an error that happens during execution of a program. When that error occurs, it terminates program execution.

Example for Syntax error:

```
a=5
b=0
```

```
print(a/b)) # Syntax Error
```

Example for Runtime Error (Exception)

```
a=5
b=0
print(a/b) # Division by Zero Exception
```

In the above example, the parser ran into an exception error. This type of error occurs whenever syntactically correct Python code results in an error.

An exception can be defined as an abnormal condition in a program. It interrupts the flow of the program. Whenever an exception occurs, the program halts the execution, and thus the further code is not executed.

**Exception Handling**

Exception handling is one of the most important feature of Python programming language that allows us to handle the errors caused by exceptions. Errors detected during execution are called exceptions.

An exception obstructs the normal flow of the program execution. When an exception occurs in the program, execution gets terminated. In such cases, python provides a system-generated error message (not easy to understand the reason behind the error). However, programmer can also handle the exceptions, by providing meaningful message to the user about the issue rather than a system-generated message.

**Exceptions are of two types:**

- Built-in exceptions
    - Python interpreter or built-in functions can generate the built-in exceptions
- User-defined exceptions.
    - User-defined exceptions are custom exceptions created by the user.

**Built-in Exceptions:**

- ZeroDivisionError: Occurs, when we try to divide a number by zero
- NameError: Occurs, when a variable/object is not declared
- TypeError: Occurs, when operands are of different type

**Exception handling by using try...except...finally**

When exceptions are handled properly, flow of the program execution does not get interrupted when an exception occurs. Handling of exceptions results in the execution of all the statements in the program.

To handle exceptions, programs are written by using try...except...finally statements.

The syntax for try...except...finally is,

```
try:
        statement_1
except Exception_Name_1:
        statement_2
except Exception_Name_2:
        statement_3
        .
        .
        .
else:
        statement_4
finally:
        statement_5
```

A **try** block consisting of one or more statements that might raise an exception. If exception is raised in the try block, control moves to except block. if no exceptions are raised in try block, except block is skipped off.

**Except** blocks are used to handle exceptions thrown by the try block. There can be more than one except blocks. Multiple except blocks with different exception names can be chained together. The except blocks are evaluated from top to bottom in your code, but only one except block is executed for each exception that is thrown.

If no except block has a matching exception name, then an except block that does not have an exception name is selected, if one is present in the code.

Instead of having multiple except blocks with multiple exception names for different exceptions, you can combine multiple exception names together separated by a comma (also called parenthesized tuples) in a single except block.

The syntax for combining multiple exception names in an except block is,

```
except (Exception_Name_1, Exception_Name_2, Exception_Name_3):
        statement(s)
```

where Exception_Name_1, Exception_Name_2 and Exception_Name_3 are different exception names.

In try...except statement, else and finally blocks are optional blocks. These optional statements must follow after all the except blocks.

- else block will be executed only if no exception is raised in try block.

- A finally block is always executed before leaving the try statement, whether an exception has occurred or not.

Note: When an exception has occurred in the try block and has not been handled by any except block, it is re-raised after the finally block has been executed.

You can also leave out the name of the exception after the except keyword.

Examples of built-in exceptions:

```
#Value error
print("Enter a number")
try:
        a= int(input())
        print(" number entered by you is ", a)
except ValueError:
        print("Please enter only numbers")
else:
        print("Proceed Further”)
finally:
        print("try was ended")
```

```
#division by zero error
print("Enter a number")
try:
  a= int(input("Enter a value"))
  b= int(input("Enter b value"))
  c = a // b
  print(" division of a and b is ", c)
except ZeroDivisionError:
  print("b value cannot be zero")
else:
  print("Proceed Further")
finally:
  print("try was ended")
```

<u>other exceptions handlers</u>

```
#check voting age using exception handling
age = int(input("enter your age"))
try:
  if age < 18:
    print(f" You entered age as {age}")
```

```
        raise
    else:
        print("You are elgible for voting in India")
except Exception:
    print("Not eligible for voting in India")
else:
    print("Use your right to vote")
finally:
    print("Came to the end of try block")
```

# Basic input and output functions of python

**Input Function:**

In Python, ***input()*** function is used to read data from the user.

The syntax for input function is:

> ***variable_name*** = ***input(["prompt"])***

The *prompt* statement gives an indication to the user of the value that needs to be entered through the keyboard. prompt is an optional one here.

Whatever the data that is read form the user or keyboard is of string type. So, we have to use type conversion function to change the string type data into required type.

**Output Function:**

To display the results to the user or to display any information on the console/screen python uses ***print()*** function.

The ***print()*** function will print everything as strings and anything that is not already a string is automatically converted into string representation. **p*rint()*** function generates a new line after printing the string.

There are different ways to print values on the console/screen in Python. Few of them are

- **str.format()**
- **f-string**

***str.format()*** **Method**

*format()* method is used to insert the value of a variable, expression or an object into another string and display it to the user as a single string. The *format()* method returns the string values.

The syntax for *format()* method is,

> ***str.format(p0, p1, ...)***
> ***str.format(k0=v0, k1=v1, ...)***

where

- p0, p1,... are called as positional arguments.
- k0, k1,... are keyword arguments with their assigned values of v0, v1,... . keys are replaced by values.

Positional arguments are a list of arguments that are accessed with an index of argument inside curly braces like {index}. Index value starts from zero.

Keyword arguments are a list of arguments of type *keyword* = *value*, that can be accessed with the name of the argument inside curly braces like {keyword}.

Example: **Program to Demonstrate the Positional Arguments**

```
a = 10
b = 20
print("The values of a is {0} and b is {1}".format(a, b))
print("The values of b is {1} and a is {0}".format(a, b))
```

Example: **Program to Demonstrate the keyword=value Arguments**

```
print("Give me {ball} ball".format(ball = "tennis"))
```

**f-Strings:**
Formatted strings or f-strings were introduced in Python 3.6. A *f-string* is a string literal that is prefixed with “f”. These strings may contain placeholders, which are expressions/variables enclosed within curly braces {}. At the time of displaying output on the console/screen expressions/variables are replaced by their respective values.

Syntax:

```
print(f"string {} string”)
```

**Example: Given the Radius, Write Python Program to Find the Area and Circumference of a Circle**

```
radius = int(input("Enter the radius of a circle"))
area_of_a_circle = 3.1415 * radius * radius
circumference_of_a_circle = 2 * 3.1415 * radius
print(f"Area = {area_of_a_circle} and Circumference = {circumference_of_a_circle}")
```