Variables

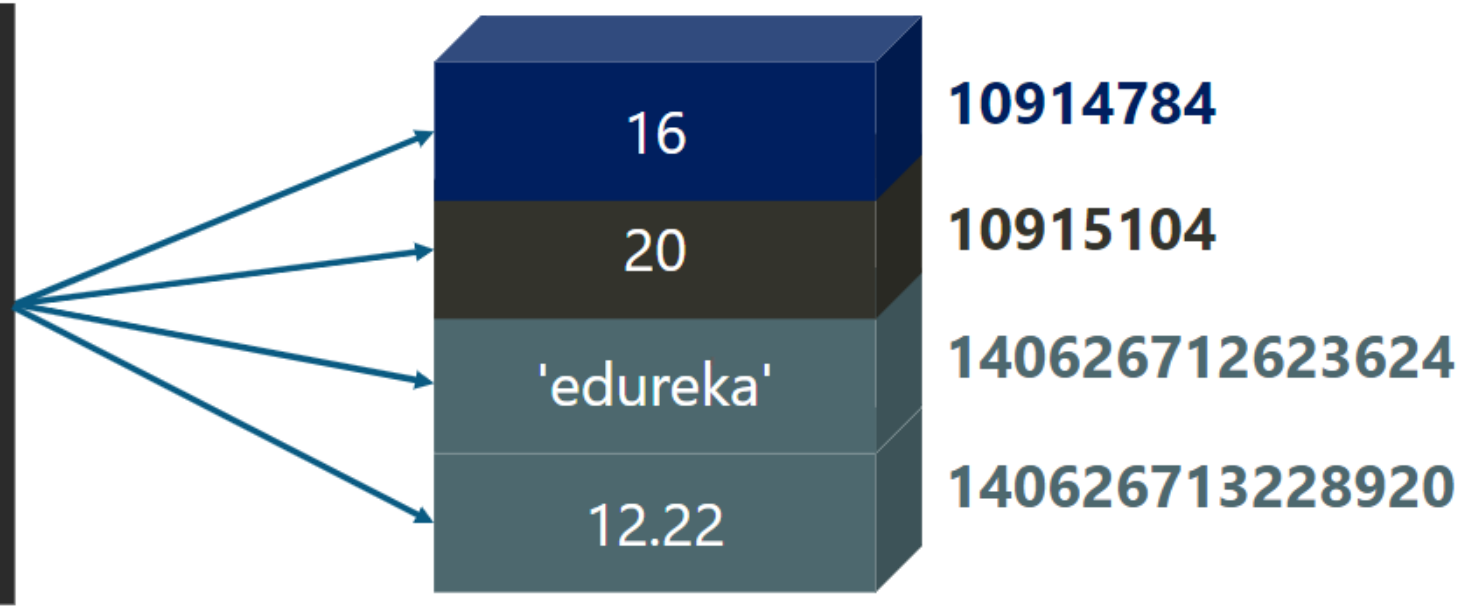
Identifier

- A Python Identifier is a name used to identify a variable, function, class, module or other objects
- An Identifier starts with a letter (A to Z or a to z) or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9)
- Python is case sensitive
- Python does not allow special characters such as @, $ and % within identifiers

Variables

Variables are reserved memory locations to store values. This means that when you create a variable, you reserve some space in memory.

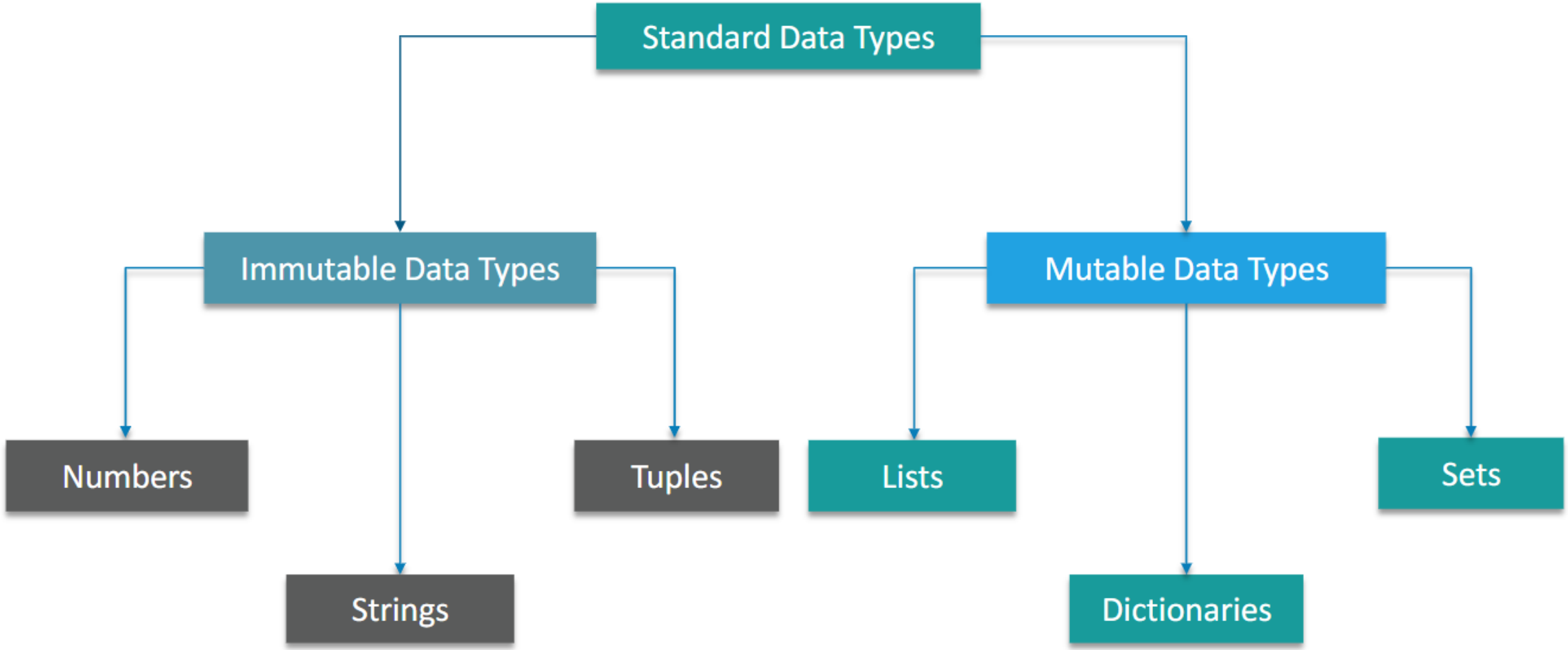
```
a = 16
id(a)
b = 20
id(b)
b ='edureka’
id(b)
b = 12.22
id(b)
```



Note: ***id()*** is a python inbuilt function which returns the unique identity of an object

Standard Data Types

Standard Data Types



Immutable and Mutable Data Types

```
In [47]: lang = 'Java'

In [48]: print('Original ID of lang variable -->',id(lang))
         Original ID of lang variable --> 1693849167280

In [49]: lang = 'Python'

In [50]: print('ID after assigning new value Python to lang variable -->',id(lang))
         ID after assigning new value Python to lang variable --> 1693841712112
```

Values of **Immutable Objects** cannot be changed

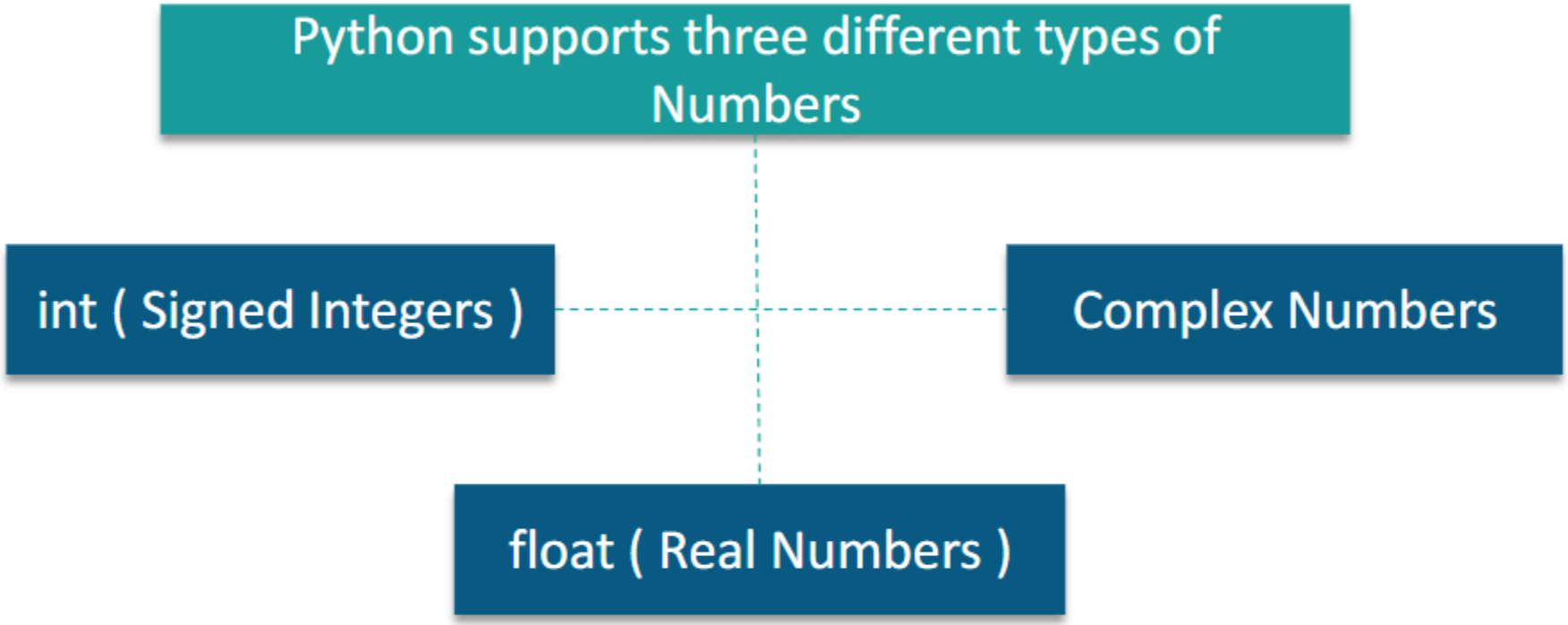
NOTE: Memory address (ID) is changed

Values of **Mutable Objects** can be changed

NOTE: Memory address (ID) remains the same

```
In [51]: lst = [1,2,3,4]

In [53]: print('Original ID of the list -->',id(lst))
         Original ID of the list --> 1693923158920

In [54]: lst.append(5)

In [55]: lst
Out[55]: [1, 2, 3, 4, 5]

In [56]: print('ID of the list after appending 5 -->',id(lst))
         ID of the list after appending 5 --> 1693923158920
```

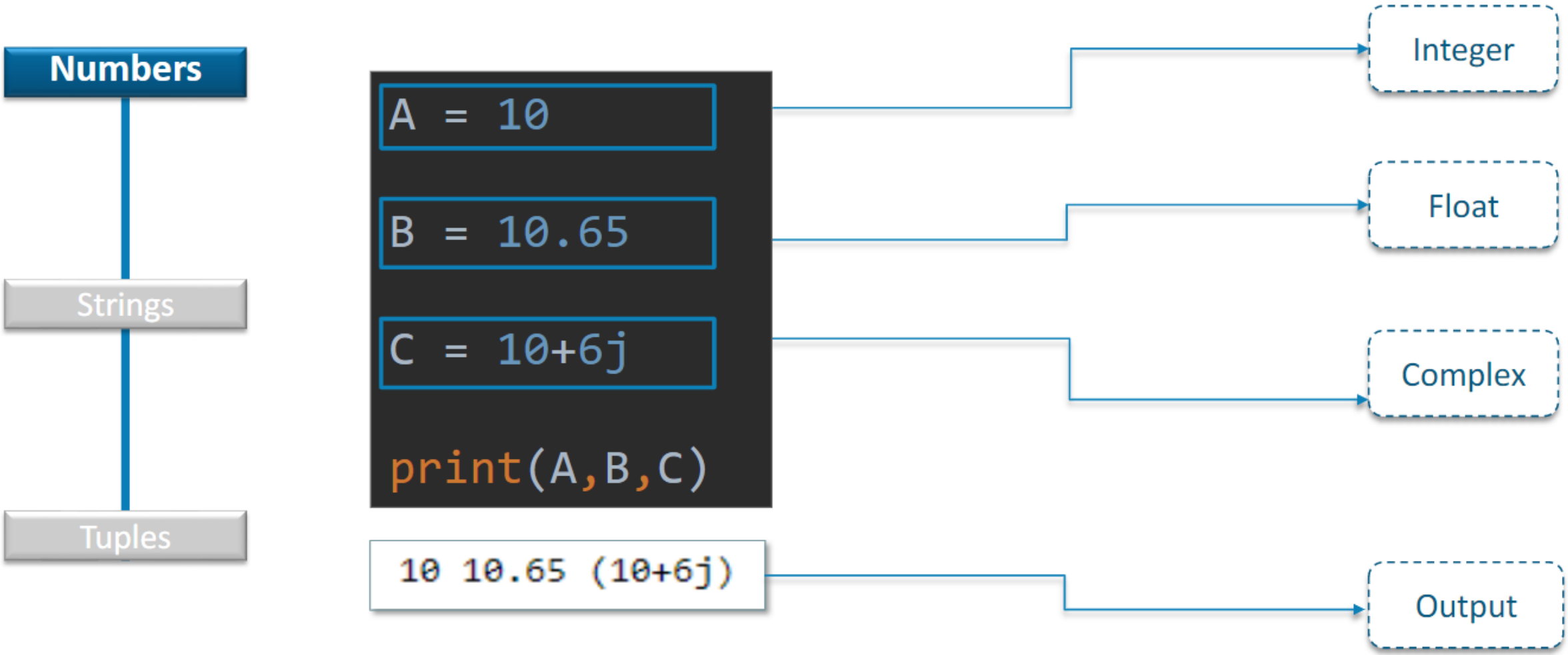
Immutable Data Types - Numbers



In Python you can represent **Numbers** in multiple ways:

- *Binary*
- *Octal*
- *Hexadecimal*

Immutable Data Types – Numbers (Cont.)



Numbers In Python – Type and Instance

We can check the type of variable or the value or the function to which class it belongs using ***type()*** and ***isinstance()*** function

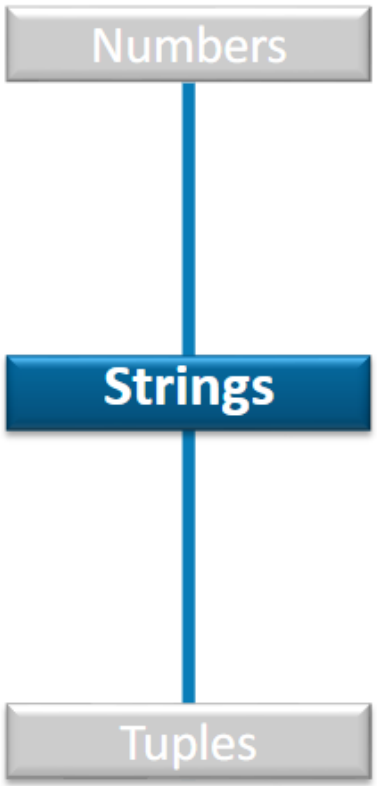
```
1  x=10
2  y=22.33
3  z=44+55j
4
5
6  #Type
7  print(type(x))
8  print(type(y))
9  print(type(z))
10
11 #Check instance
12 print(isinstance(x,int))
13 print(isinstance(x,float))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
True
False
```

Output

Demo 1: Numbers in Python

Immutable Data Types - Strings



- **Strings** are sequence of characters represented within quotes
- Characters in python are treated as strings of length one

```
A = 'Welcome To edureka!'
B = "Python is Great"
C = ''' High
          level '''
D = """Programming
          Language"""
print(A,B,C,D)
```

Different ways of defining a string

```
Welcome To edureka! Python is Great High
level Programming
Language
```

Output

Immutable Data Types - Tuples

Numbers

Strings

Tuples

Tuple is a collection of objects separated by comma enclosed within parenthesis.

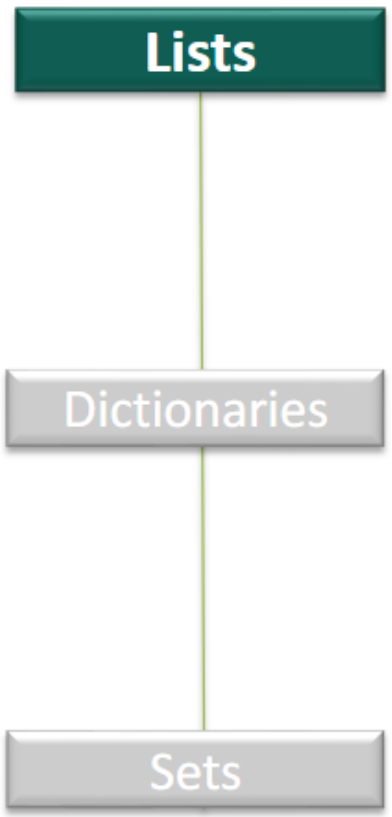
```
A=(1,2,3.15,'edureka!')

print(A)
```

A tuple can have objects of different data types, unlike arrays in **'C'**

```
(1, 2, 3.15, 'edureka!')
```

Output

Mutable Data Types - Lists



List is a collection of objects separated by comma enclosed within square brackets.

```
A=[1,2,3.15,'edureka!']

print(A)
```

Lists are enclosed within square brackets

```
[1, 2, 3.15, 'edureka!']
```

Output

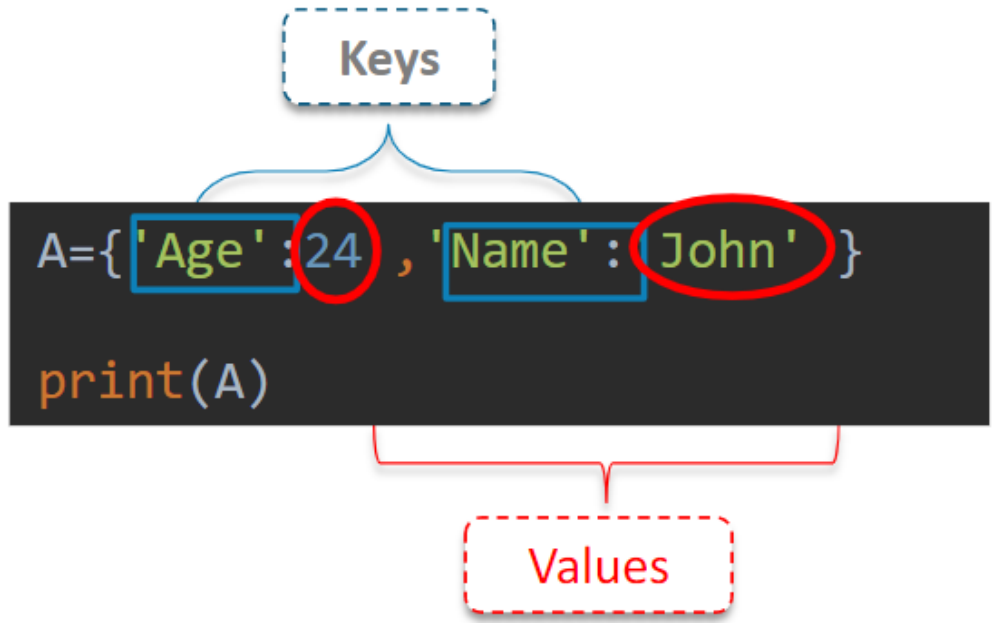
Mutable Data Types - Dictionaries

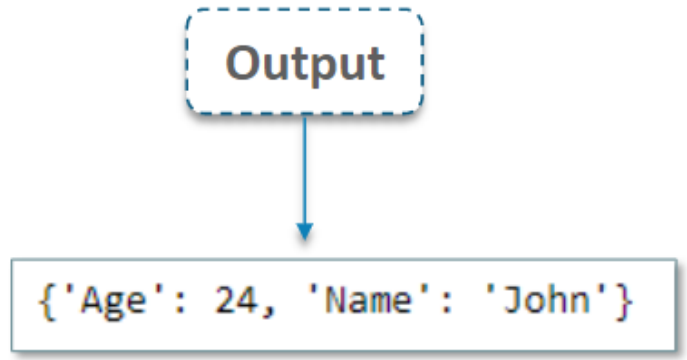
Lists

Dictionaries

Sets

- Dictionaries are a collection of objects as a key value pair
- Each key and its value is separated by a colon (:)
- The key-value pair are separated by commas
- All the key-value pairs are enclosed within curly braces





Mutable Data Types - Sets

Lists

Dictionaries

Sets

Collection of unordered unique items inside curly braces {}, separated by comma.

Every element in a set must be unique

```
A={1,2,3,3}

print(A)
```

Output – Notice that 3 has appeared only once

```
{1, 2, 3}
```

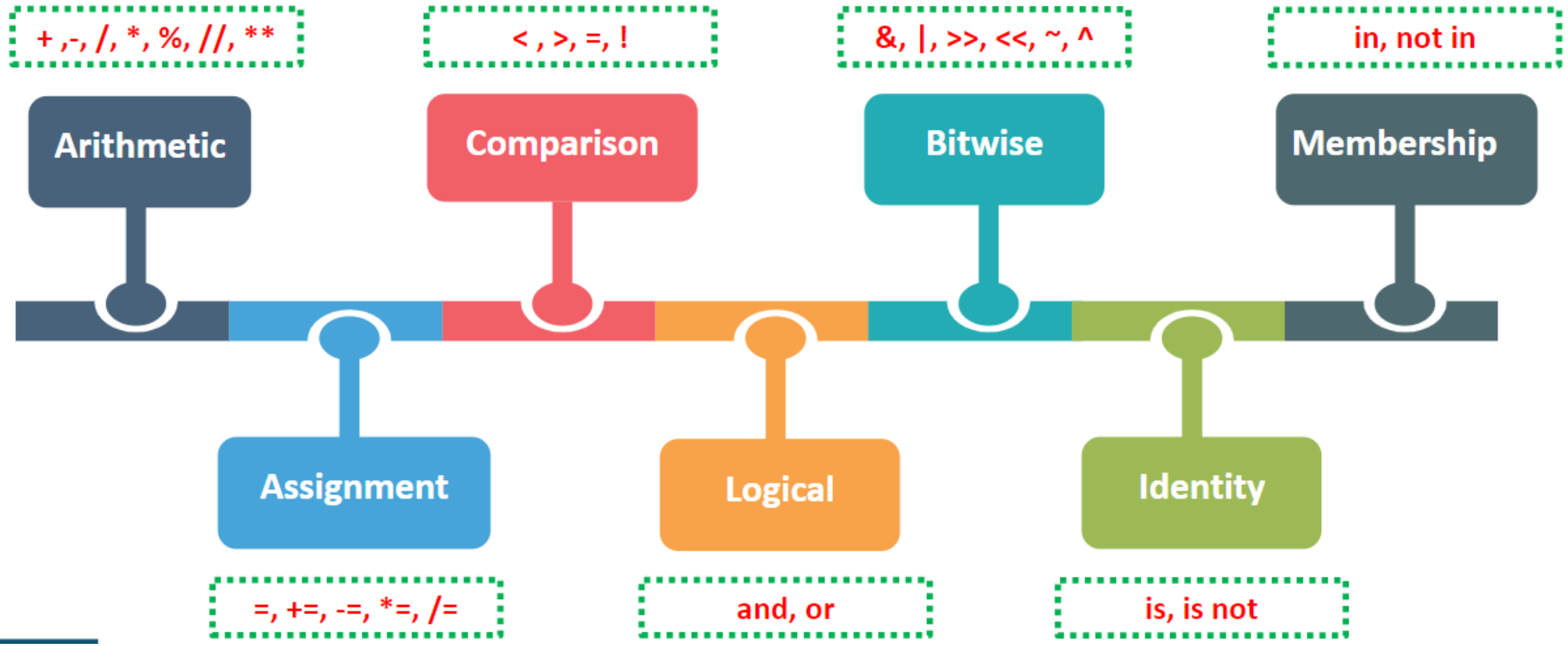
You can also create a set by calling a built-in function ***set()***

Python Operators

Operators in Python

Operators perform operations on the Operands.

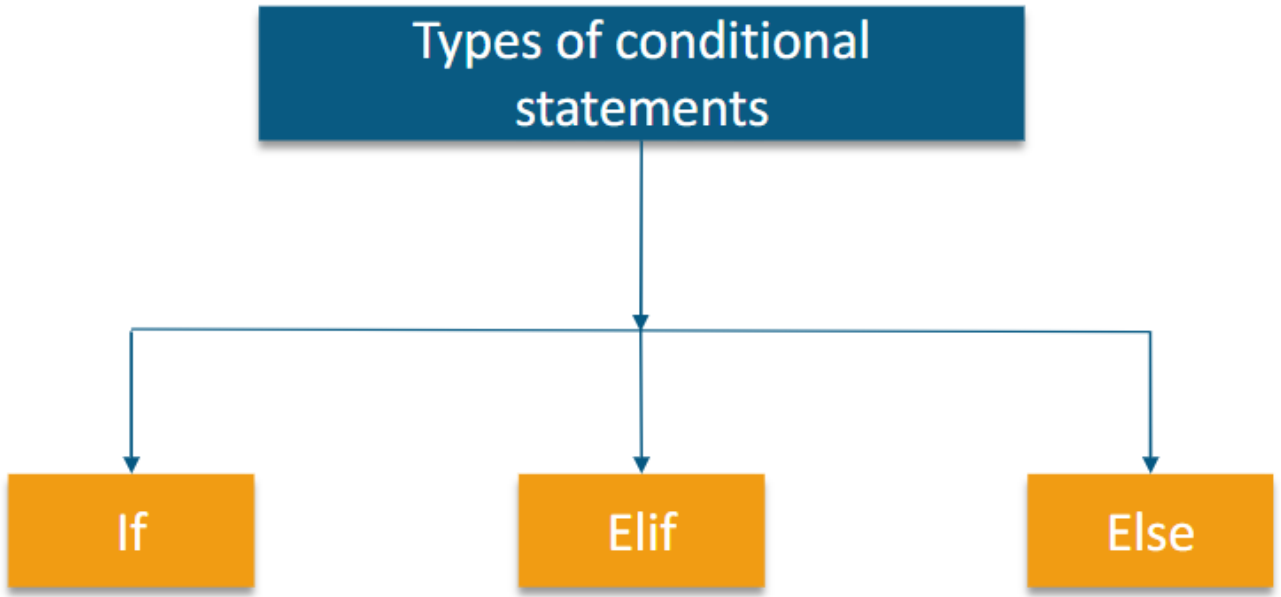
For example, 2 + 3 = 5, here **2** and **3** are **Operands** and **+** is called **Operator**.

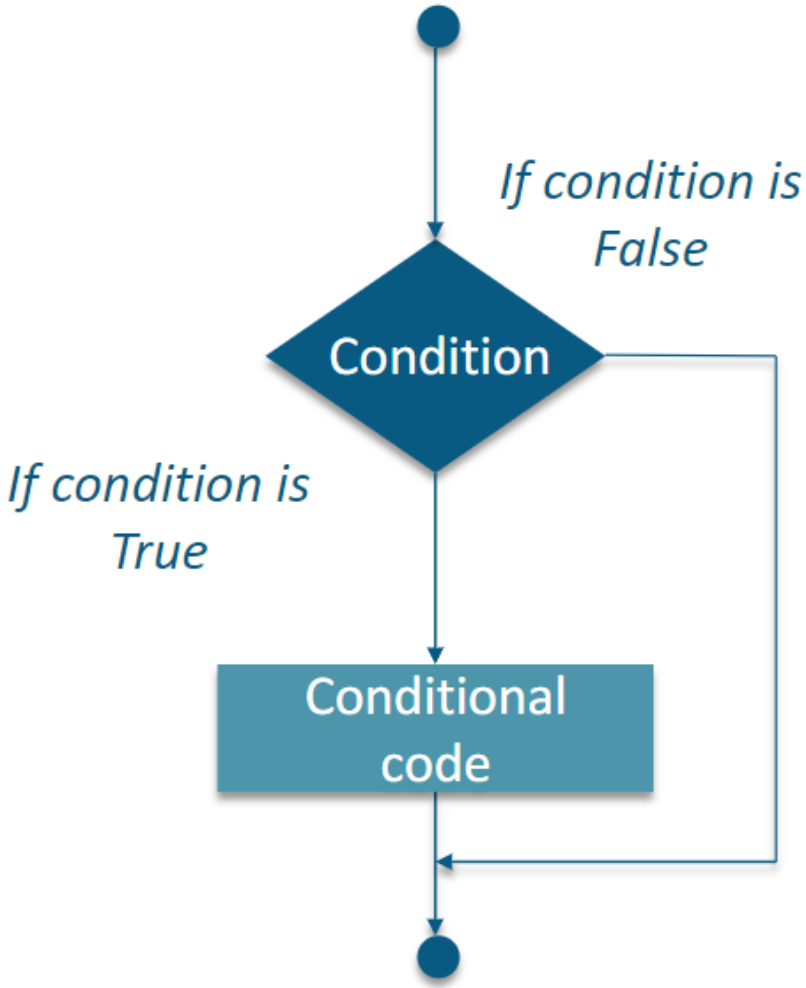


Control Structures- Conditional Statements

Conditional Statements

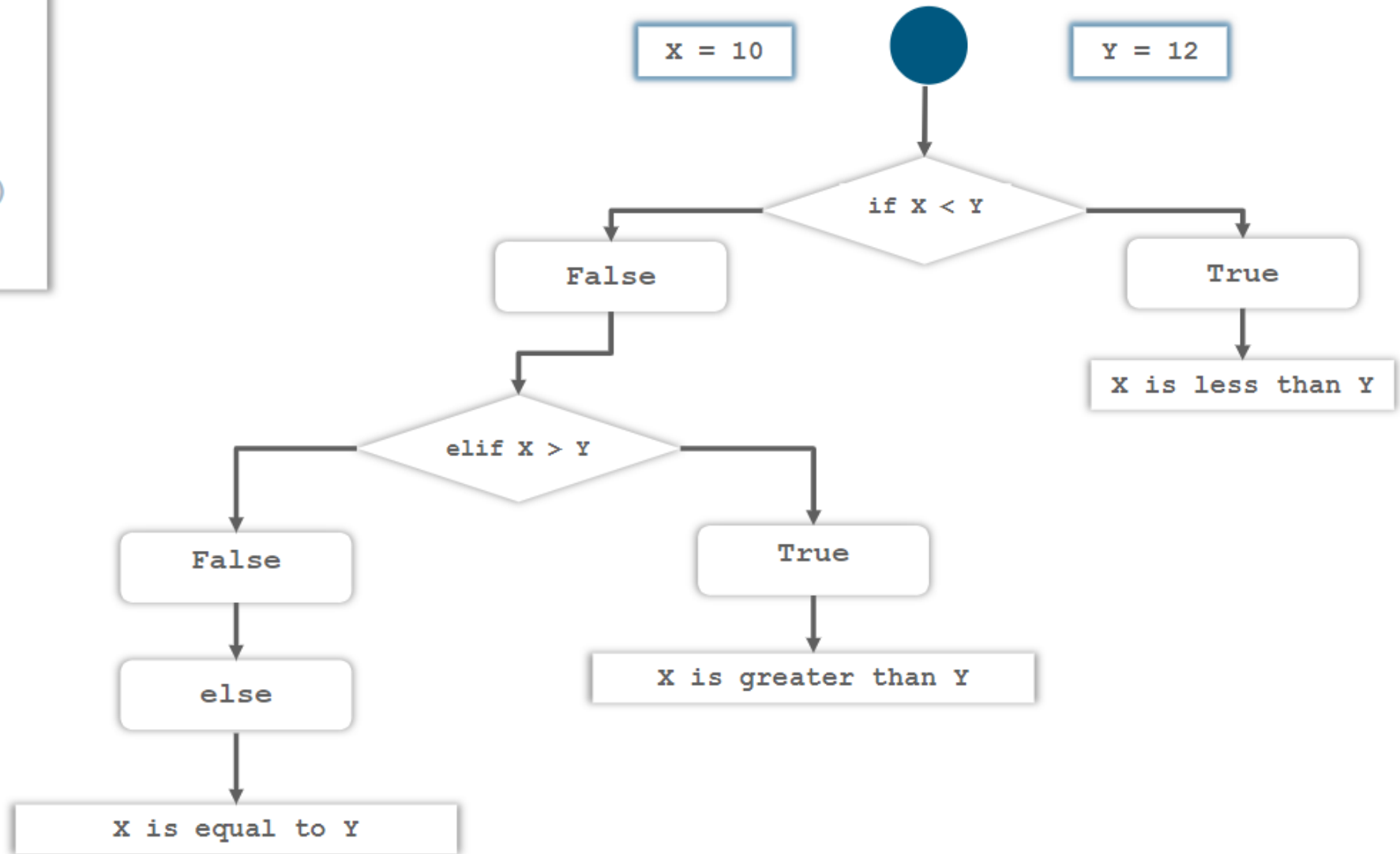
Conditional statements are used to execute a statement or a group of statements when some condition is **True**





Conditional Statements: `if`, `elif`, and `else` contd.

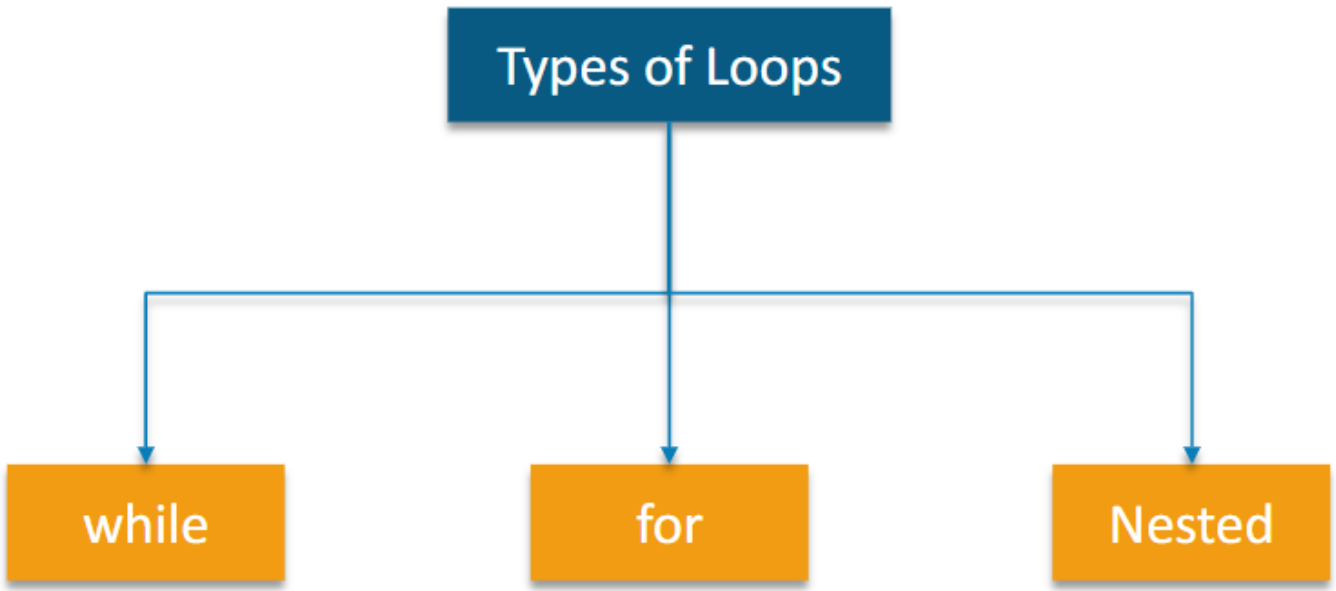
```
X = 10
Y = 12
if X<Y:
    print('X is less than Y')
elif X>Y:
    print('X is greater than Y')
else:
    print('X is equal to Y')
```

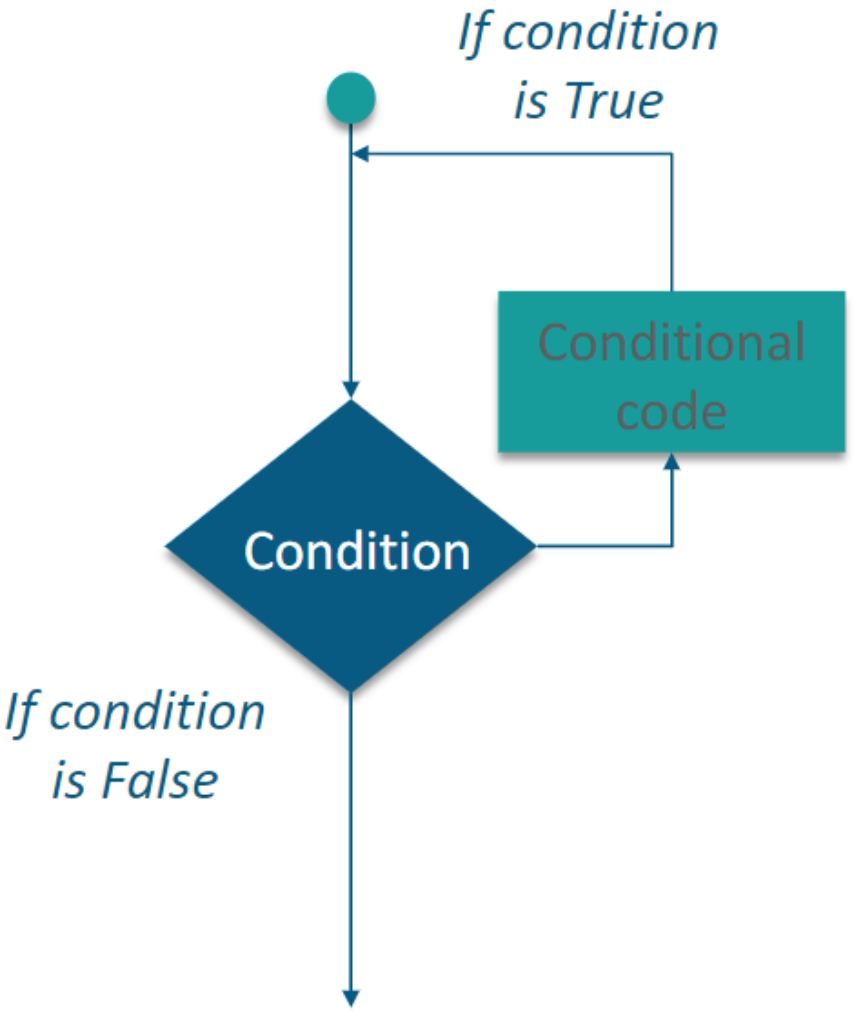


Control Structures- Loops

Loops

A loop statement allows us to execute a statement or a group of statements multiple times



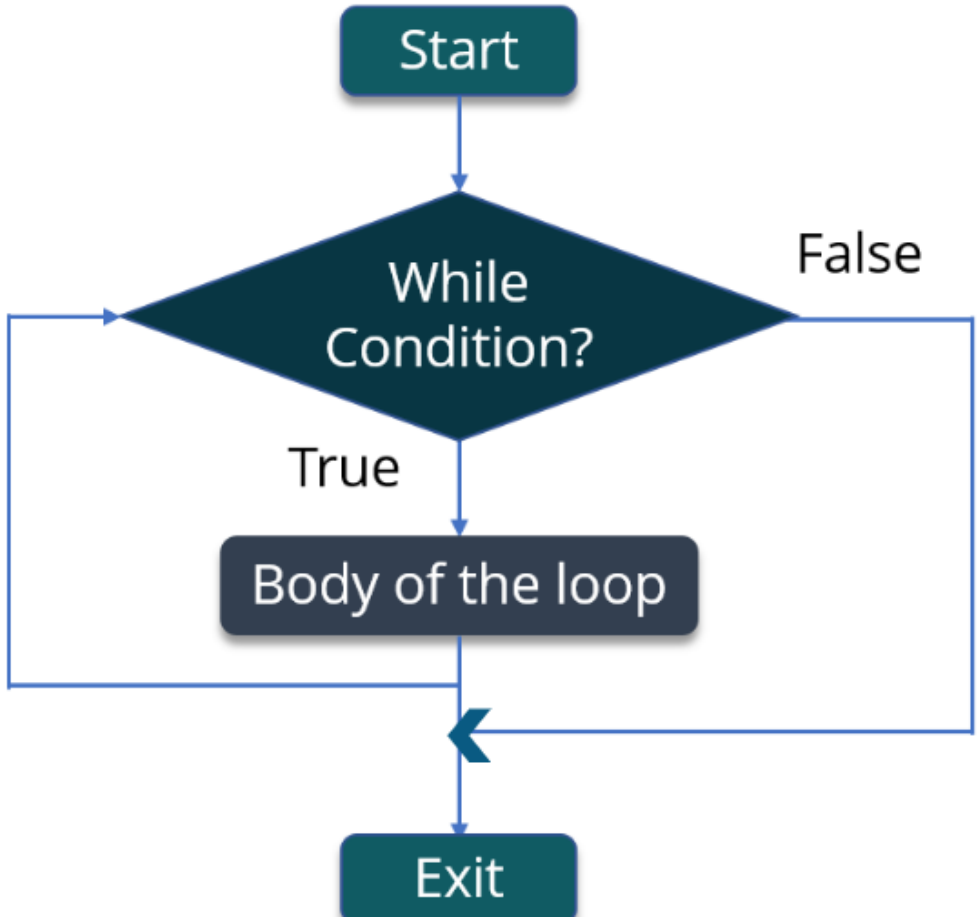


while Loop

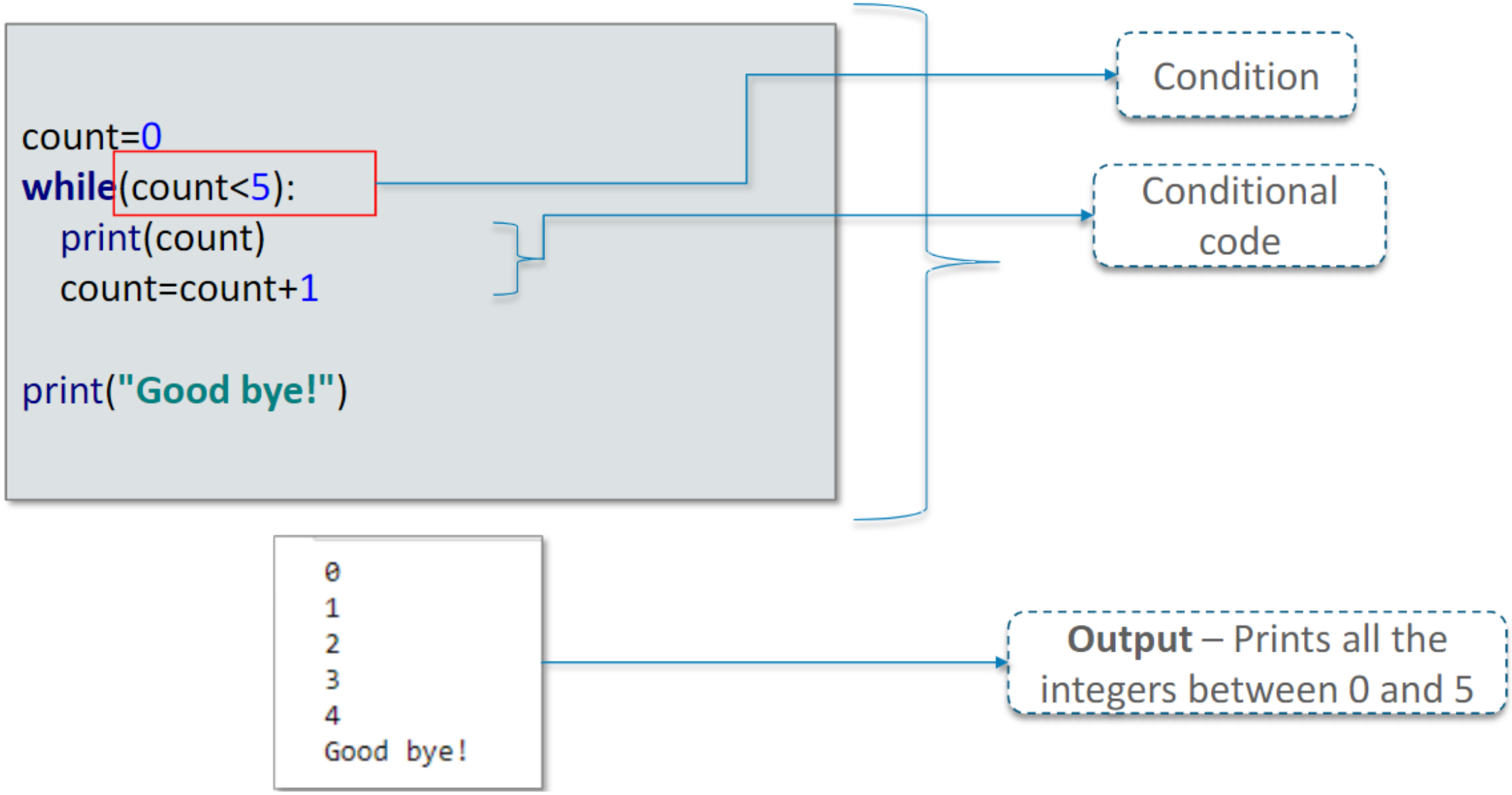
while loops keep on iterating the statement/block until certain conditions are met

Syntax :

```
while expression:
    statements
```



while Loop Example

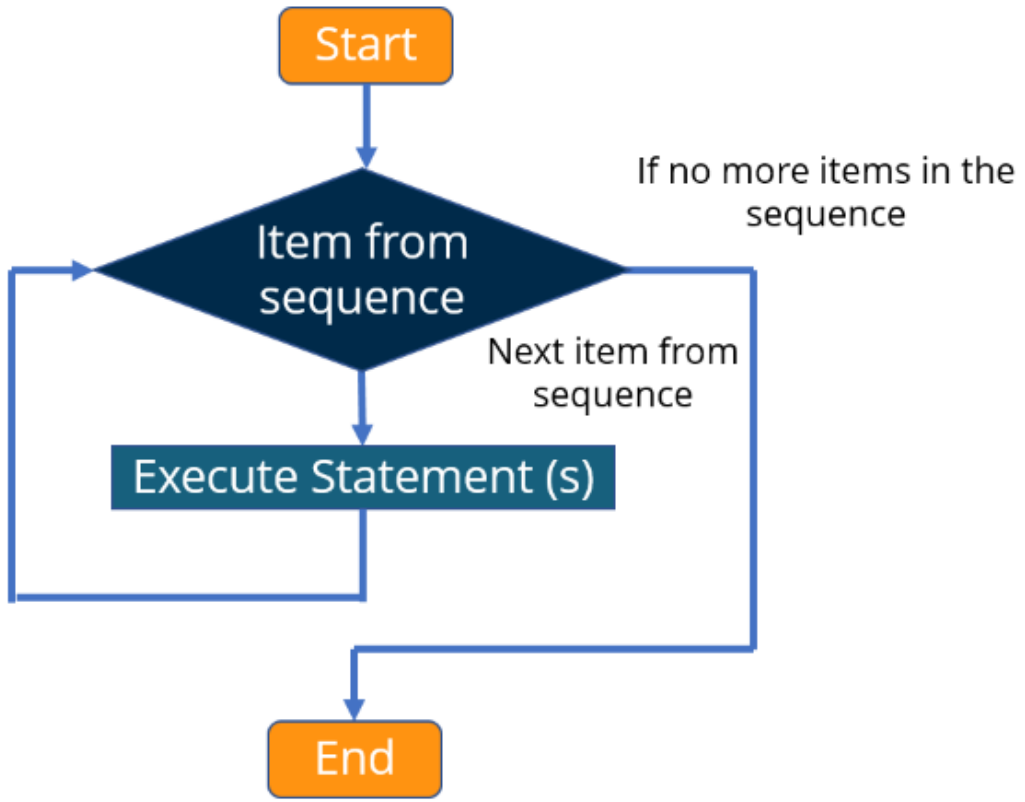


for Loop

for is a conditional iterative statement used to execute the *block* for a specified number of times(based on the elements in range or sequence).

Syntax :

```
for <variable> in <range>:
    stmt1
    stmt2
    ...
    stmtn
```



for Loop Example

Please note that, the stop index is not included in the result of *for* loop

```
for i in range(1,5):
    print("Welcome to for loop",i)
```

Output

```
Welcome to for loop 1
Welcome to for loop 2
Welcome to for loop 3
Welcome to for loop 4
```

```
for i in range(5,1,-1):
    print("Welcome to for loop",i)
```

Output

```
Welcome to for loop 5
Welcome to for loop 4
Welcome to for loop 3
Welcome to for loop 2
```

for-else Concept

When a ***for*** loop successfully executes without any break statement execution, then the ***else*** block attached with the ***for*** loop gets executed as well. But, if the ***for*** loop encounters break during iteration, the ***else*** part won't execute.

```
for i in range(1,5):
    print('Welcome to Edureka',i)
    if(i>=5):
        break
else:
    print('The for loop was successfully executed')
```

```
Welcome to Edureka 1
Welcome to Edureka 2
Welcome to Edureka 3
Welcome to Edureka 4
The for loop was successfully executed
```

Vs.

```
for i in range(1,5):
    print('Welcome to Edureka',i)
    if(i>=4):
        break
else:
    print('The for loop was successfully executed')
```

```
Welcome to Edureka 1
Welcome to Edureka 2
Welcome to Edureka 3
Welcome to Edureka 4
```

Nested Loops

Nested loop, basically means a loop inside a loop. It can be a ***for*** loop inside a ***while*** loop and vice-versa. It can also be a ***while*** loop inside a ***while*** loop or ***for*** loop inside a ***for*** loop.

```
count=1
for i in range(10):
    print(str(i)*i)

    for j in range(0,i):
        count=count+1
```

for loop inside a ***for*** loop

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Output

Loop Control Statements

- Loop control statements are used to alter the execution flow from its normal flow.

Control Statement	Description
break statement	Is used to terminate the loop and the execution flow goes to the statement immediately following the loop
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating
pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute

Loop Control Statements- Example

```
for i in range(10,50):
    print(i)
    if(i==30):
        break
```

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

```
for j in range(1,11):
    print(j)
    if(j==5):
        continue
```

```
1
2
3
4
5
6
7
8
9
10
```

```
for k in range(1,3):
    pass
print("Loop ends here")
```

```
Loop ends here
```

Demo 2: Conditional Statements and Loops

Command Line Parameters

Command Line Parameters

- It is possible to pass **arguments** to Python programs when they are executed
- The brackets which follow main are used for this purpose
- **. argv** refers to the number of **arguments** passed, and **argv[]** is a pointer array which points to each **argument** which is passed to main
- The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes:
  - **sys.argv** is the list of command-line arguments
  - **len(sys.argv)** is the number of command-line arguments

Command Line Parameters- Example

- **Example:**
  - Consider the following script test.py:

```
#!/usr/bin/python
import sys
print ('Number of arguments:',
len(sys.argv), 'arguments.' )
print ('Argument List:', str(sys.argv) )
```

  - Now, run above script as follow:

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

  - After running this script, output will be:

```
Number of arguments: 4 arguments. Argument
List: ['test.py', 'arg1', 'arg2', 'arg3']
```

Reading Keyboard Input

Reading Keyboard Input

Python provides a built-in function ***input*** () to read a line of text from the standard function

```
user_input=input('Enter Your value')
print('The value entered by user:',user_input)
print('The datatype of the value entered by the user:',type(user_input))
```

Output

```
Enter Your value10
The value entered by user: 10
The datatype of the value entered by the user: <class 'str'>
```

Although the user entered an integer, the data type shown is string. How is it possible? Is the interpreter working right?

Reading Keyboard Input – *eval()* Function

Reading Keyboard Input

By default, all the inputs entered by users are considered as **string**. Python provides a built-in function ***eval()*** to retain the original data type of the entered value

```
user_input=input('Enter Your value')
print('The value entered by user:',user_input)
print('The datatype of the value entered by the user:',type(eval(user_input)))
```

Output

```
Enter Your value10
The value entered by user: 10
The datatype of the value entered by the user: <class 'int'>
```

Do we have any other method to get back the original data type?

```
user_input=int(input('Enter Your value'))
print('The value entered by user:',user_input)
print('The datatype of the value entered by the user:',type(user_input))
```

Opening and Closing Files

Before reading and writing any data into a file, it is important to learn how to open and close a file

Opening files

Unless you open a file, you can not write anything in a file or read anything from it

Closing files

Once you are done with reading or writing, close the file

open() Function

- You can open Files using Python's built-in *open()* function

```
file_Object=open(file_name,[access_mode])
```

- Here are the parameter details:

  file_name: The file_name argument is a **string** value that contains the name of the file that you want to access

  access_mode: The access_mode determines the mode in which the file has to be opened, i.e., read, write, append etc.

open() Function – Access Modes

Modes	Description
r	This is the default mode and is used for opening a file in read only mode
rb	opens a file to read only in binary form
r+	opens a file for both reading and writing
rb+	opens a file to read and write in binary format
w	opens a file in write only mode. If the file exists, it overwrites the same or else creates a new one.
wb	opens a file for writing only in binary format. If the file exists, it overwrites the same or else creates a new one.

open() Function – Access modes (Cont.)

Modes	Description
a	opens a file to append
ab	opens a file to append in binary format
a+	opens a file to append and read
ab+	opens a file to append and read in binary format
w+	opens a file to read and write
wb+	opens a file to read and write in binary format

Writing Files

fileObject.write(string)

The ***write()*** method does not add a newline character **'\n'** to the end of the string

The ***write()*** method writes content in an open file.

Note:- Python strings can have binary data and not just text

Reading Files

fileObject.read([count])

The ***read()*** method reads a string from an open file

Note :- It is important to note that Python strings can have binary data apart from text data

Renaming Files

os.rename(current_file_name, new_file_name)

The ***rename()*** method takes two arguments, the **current filename** and the **new filename**

rename() is the method from ***os*** module. We are going to learn ***os*** module in detail in **Module 4**

Deleting Files

```
os.remove(file_name)
```

You can use the ***remove()*** method to delete files by supplying the name of the file to be deleted as an argument

remove() is the method from ***os*** module

Closing Files

file.close() → The ***close()*** method closes the opened file

Note :- A closed file cannot be read or written any more

Note :- Python automatically closes a file when the reference object of a file is reassigned to another file

Demo : User Input and File Handling