

PYTHON PROGRAMMING

PYTHON

- **PYTHON** is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language.
- Python is dynamically-typed and garbage-collected programming language.
- It was created by **Guido van Rossum** during 1985- 1990.
- Python supports multiple programming paradigms, including Procedural, Object Oriented and Functional programming language. Python design philosophy emphasizes code readability with the use of significant indentation.

Why to Learn Python?

There are many other good reasons which makes Python as the top choice of any programmer:

- Python is Open Source which means its available free of cost.
- Python is simple and so easy to learn
- Python is versatile and can be used to create many different things.
- Python has powerful development libraries include AI, ML etc.
- Python is much in demand and ensures high salary

Advantages of learning Python

- Easy to read, learn and code
- Dynamic Typing
- Free, Open Source
- Portable
- Extensive Third-Party Libraries
- Wide Range of Applications
- Extensible and Integrable to Other Programming Languages
- Memory Management
- Improved Productivity
- Involvement in Large Projects

Disadvantages of Python

- **It's Simple Nature**
- **Slow Speed and Memory Inefficient**
- **Weak Mobile Computation** Python has many applications on the server-side. But it is hardly seen on the client-side or in mobile applications. The main reasons for this are:
 - It occupies a lot of memory.
 - This makes the process slow.
 - It also does not facilitate security
- **Poor Database Access** Python databases are weak compared to other technologies like JDBC (Java Data Base Connectivity) and ODBC (Open Data Base Connectivity). This limits its use in high enterprises.
- **Runtime Errors due to dynamic typing** Because it's a dynamically typed language, you do not need to declare any variable and a variable storing an integer can store a string later. This might happen unnoticeably but leads to runtime errors while doing operations.

Python Syntax

The Python syntax defines a set of rules that are used to create Python statements while writing a Python Program.

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python Keywords

- Keywords are predefined, reserved words used in Python programming that have special meanings to the compiler.
- We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the Python language.
- All the keywords except True, False and None are in lowercase and they must be written as they are.
- Python contains thirty-five keywords in the most recent version, i.e., Python 3.8.

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

Python Identifiers

Identifiers are the name given to variables, classes, methods, etc

Rules for Naming an Identifier

1. Identifiers cannot be a keyword.
2. Identifiers are case-sensitive.
3. It can have a sequence of letters and digits. However, it must begin with a letter or `_`. The first letter of an identifier cannot be a digit.
4. It's a convention to start an identifier with a letter rather `_`.
5. Whitespaces are not allowed.
6. We cannot use special symbols like `!`, `@`, `#`, `$`, and so on.

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code

There are three types of comments available in Python

- Single line Comments
- Multiline Comments
- Docstring Comments

1. Single line Comments

- A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line.
- If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation.

EX: # This is a single line comment in python

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

2. Multiline Comments

- Python does not provide a direct way to comment multiple line. You can comment multiple lines as follows

EX: `#This is comment`

`#This is comment, too`

`#This is comment, too`

- Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:
- Another way of doing this is to use triple quotes, either `'''` or `"""`.

EX: `''' This is`

`a multiline`

`comment. '''`

`print ("Hello, World!")`

3. Docstring Comments

- Python docstrings provide a convenient way to provide a help documentation with Python modules, functions, classes, and methods.
- The **docstring** is then made available via the `__doc__` attribute.

EX: `def add(a, b):`
 `"""Function to add the value of a and b"""`
 `return a+b`

`print(add.__doc__)`

The result is: `Function to add the value of a and b`

Python Indentation

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

Python Variables

- Python variables are the reserved memory locations used to store values within a Python Program. This means that when you create a variable you reserve some space in the memory.
- Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.
- Variables are containers for storing data values.

EX: `x = 50`

`y = 59.7`

`z = "GTTC"`

Python Variable Names

- Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like color, age, name etc.
- There are certain rules which should be taken care while naming a Python variable:
 - A variable name must start with a letter or the underscore character
 - A variable name cannot start with a number or any special character like \$, (, * % etc.
 - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
 - Python variable names are case-sensitive which means Name and NAME are two different variables in Python.
 - Python reserved keywords cannot be used naming the variable.

EX: counter = 1000

_count = 100

name1 = "GTTC"

name2 = "IoT"

Age = 20

net_salary = 100000

print (counter)

print (_count)

print (name1)

print (name2)

print (Age)

print (net_salary)

RESULT: 1000

100

GTTC

IoT

20

100000

Multi Words Variable Names

- Variable names with more than one word can be difficult to read.
- There are several techniques you can use to make them more readable:
 - 1. Camel Case**

Each word, except the first, starts with a capital letter:
EX: `internetOfThings = IoT`
 - 2. Pascal Case**

Each word starts with a capital letter:
EX: `InternetOfThings = IoT`
 - 3. Snake Case**

Each word is separated by an underscore character:
EX: `internet_of_things = IoT`

Python Basic Input and Output

Python Output

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

Syntax of `print()`:

The python print function accepts **5** parameters

`print(object= separator= end= file= flush=)`

Here,

- `object` - value(s) to be printed
- `sep` (optional) - allows us to separate multiple objects inside `print()`.
- `end` (optional) - allows us to add add specific values like new line `"\n"`, tab `"\t"`
- `file` (optional) - where the values are printed. It's default value is `sys.stdout` (screen)
- `flush` (optional) - boolean specifying if the output is flushed or buffered. Default: `False`

Example 1: Python Print Statement

```
print('Good Morning!')  
print('It is rainy today')
```

Output

```
Good Morning!  
It is rainy today
```

- In the above example, the print() statement only includes the object to be printed. Here, the value for end is not used. Hence, it takes the default value '\n'.

So we get the output in two different lines.

Example 2: Python print() with end Parameter

```
# print with end whitespace  
print('Good Morning!', end= ' ')  
print('It is rainy today')
```

Output

Good Morning! It is rainy today

- Here we have included the end= ' ' after the end of the first print() statement.

Hence, we get the output in a single line separated by space.

Example 3: Python print() with sep parameter

```
print('New Year', 2023, 'See you soon!', sep= ' . ')
```

Output

New Year. 2023. See you soon!

In the above example, the print() statement includes multiple items separated by a comma.

Notice that we have used the optional parameter sep= ". " inside the print() statement.

Hence, the output includes items separated by . not comma.

Example 4: Print Python Variables and Literals

We can also use the `print()` function to print Python variables.

For example,

```
number = -10.6  
name = "GTTC Hubli"  
# print literals  
print(5)  
# print variables  
print(number)  
print(name)
```

Output

```
5  
-10.6  
GTTC Hubli
```

Example 5: Print Concatenated Strings

We can also join two strings together inside the `print()` statement. For example,

```
print('Internet ' + 'of ' + 'Things.')
```

Output

Internet of Things.

- Here, the `+` operator joins two strings `'Internet'`, `'of'`, and `'Things.'` the `print()` function prints the joined string

Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the `str.format()` method.

For example,

```
x = 5
```

```
y = 10
```

```
print('The value of x is { } and y is { }'.format(x,y))
```

- Here, the curly braces { } are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

Python Input

While programming, we might want to take the input from the user. In Python, we can use the `input()` function.

Syntax of `input()`

`input(prompt)`

Here, prompt is the string we wish to display on the screen. It is optional.

Example: Python User Input

```
# using input() to take user input  
num = input('Enter a number: ')  
print('You Entered:', num)  
print('Data type of num:', type(num))
```

Output

```
Enter a number: 10  
You Entered: 10  
Data type of num: <class 'str'>
```

- In the above example, we have used the input() function to take input from the user and stored the user input in the num variable.
- It is important to note that the entered value **10 is a string**, not a number. So, type(num) returns **<class 'str'>**.
- To convert user input into a number we can use int() or float() functions as:

num = int(input('Enter a number: '))

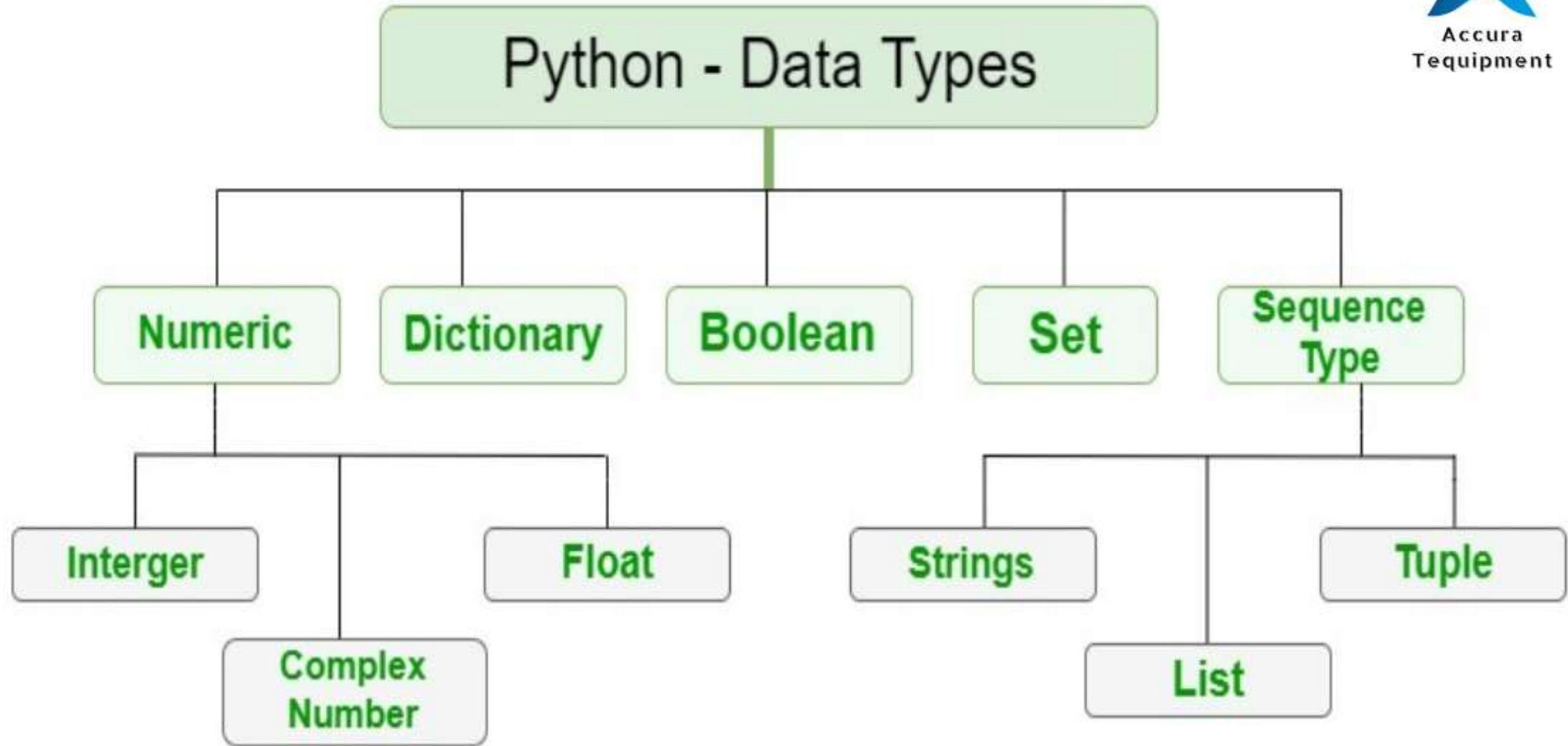
Here, the data type of the user input is converted from string to integer .

Python - Data Types

- Python Data Types are used to define the type of a variable.
- It defines what type of data we are going to store in a variable.
- The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various built-in data types, They are

- Numeric - int, float, complex
- String - str
- Sequence - list, tuple, range
- Binary - bytes, bytearray, memoryview
- Mapping - dict
- Boolean - bool
- Set - set, frozenset
- None - NoneType



Python Numeric Data Type

- Python numeric data types store numeric values. Number objects are created when you assign a value to them.

EX: $a = 55$

$b = 39.5$

$c = 1 + 5j$

- Python supports four different numerical types –
 - **int** (signed integers)
 - **long** (long integers, they can also be represented in octal and hexadecimal)
 - **float** (floating point real values)
 - **complex** (complex numbers)

Python String Data Type

- Python Strings are identified as a contiguous set of characters represented in the quotation marks.
- Python allows for either pairs of single or double quotes.
- Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator in Python.

```
str = 'Hello World!'
print (str) # Prints complete string
print (str[0]) # Prints first character of the string
print (str[2:5]) # Prints characters starting from 3rd to 5th
print (str[2:]) # Prints string starting from 3rd character
print (str * 2) # Prints string two times
print (str + "TEST") # Prints concatenated string
```

RESULT:

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```


Methods of Python String

Methods	Description
<code>upper()</code>	converts the string to uppercase
<code>lower()</code>	converts the string to lowercase
<code>partition()</code>	returns a tuple
<code>replace()</code>	replaces substring inside
<code>find()</code>	returns the index of first occurrence of substring
<code>rstrip()</code>	removes trailing characters
<code>split()</code>	splits string from left
<code>startswith()</code>	checks if string starts with the specified string
<code>isnumeric()</code>	checks numeric characters
<code>index()</code>	returns index of substring

Python List Data Type

- Python Lists are the most versatile compound data types.
- Lists are used to store multiple items in a single variable.
- List items are ordered, changeable, and allow duplicate values
- List items are indexed, the first item has index [0], the second item has index [1], etc
- A Python list contains items separated by commas and enclosed within square brackets ([]).
- Python list can be of different data type.
- The values stored in a Python list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.
- The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  
tinylist = [123, 'john']  
print (list)    # Prints complete list  
print (list[0]) # Prints first element of the list  
print (list[1:3]) # Prints elements starting from 2nd till 3rd  
print (list[2:]) # Prints elements starting from 3rd element  
print (tinylist * 2) # Prints list two times  
print (list + tinylist) # Prints concatenated lists
```

RESULT:

```
['abcd', 786, 2.23, 'john', 70.2]  
'abcd'  
[786, 2.23]  
[ 2.23, 'john', 70.2]  
[123, 'john', 123, 'john']  
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

List Methods

Method	Description
<code>append()</code>	add an item to the end of the list
<code>extend()</code>	add all the items of an iterable to the end of the list
<code>insert()</code>	inserts an item at the specified index
<code>remove()</code>	removes item present at the given index
<code>pop()</code>	returns and removes item present at the given index
<code>clear()</code>	removes all items from the list
<code>index()</code>	returns the index of the first matched item
<code>count()</code>	returns the count of the specified item in the list
<code>sort()</code>	sort the list in ascending/descending order
<code>reverse()</code>	reverses the item of the list
<code>copy()</code>	returns the shallow copy of the list

Advantages of Lists:

- **Dynamic Size :** Lists in Python are dynamic, meaning they can grow or shrink in size during runtime. Elements can be added or removed without specifying the size beforehand. This makes lists very flexible for handling varying amounts of data.
- **Mutable :** Lists are mutable, which means you can modify their elements after creation. You can change, add, or remove items in a list.
- **Versatility :** Lists can hold elements of different data types. You can have integers, strings, objects, or even other lists within a single list.
- **Rich in Built-in Functions :** Lists come with a variety of built-in functions and methods for manipulation, such as `append()`, `extend()`, `remove()`, `pop()`, and more. These functions make it easy to work with lists.
- **Ease of Use :** Lists are easy to create and manipulate in Python, making them a popular choice for many programming tasks.

Disadvantages of Lists:

- **Slower Operations** : Compared to arrays, lists in Python can be slower for certain operations. This is because lists are implemented as dynamic arrays, which sometimes require resizing and copying elements to accommodate new additions. This resizing operation can be time-consuming.
- **Higher Memory Consumption** : Lists can consume more memory than arrays or tuples because of their dynamic and versatile nature. They need extra space to accommodate potential resizing and storing different types of objects.
- **Less Efficient for Numerical Computations** : If you're working with numerical computations, arrays provided by libraries like NumPy are more efficient than lists. NumPy arrays are implemented in C and provide a significant performance boost for numerical operations.

Python Tuple Data Type

- Python tuple is another sequence data type that is similar to a list.
- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and unchangeable.
- Tuple items are ordered, unchangeable, and allow duplicate values.
- Unchangeable: Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- A Python tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated.
- Tuples can be thought of as **read-only** lists.

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )  
tinytuple = (123, 'john')  
print (tuple) # Prints the complete tuple  
print (tuple[0]) # Prints first element of the tuple  
print (tuple[1:3]) # Prints elements of the tuple starting from 2nd till 3rd  
print (tuple[2:]) # Prints elements of the tuple starting from 3rd element  
print (tinytuple * 2) # Prints the contents of the tuple twice  
print (tuple + tinytuple) # Prints concatenated tuples
```

RESULT:

```
('abcd', 786, 2.23, 'john', 70.2)  
'abcd'  
(786, 2.23)  
(2.23, 'john', 70.2)  
(123, 'john', 123, 'john')  
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```


Python Tuple Methods

There are only two tuple methods `count()` and `index()` that a tuple object can call.

- **Python Tuple `count()`**

returns count of the element in the tuple

- **Python Tuple `index()`**

returns the index of the element in the tuple

Count() Method

The count() method of Tuple returns the number of times the given element appears in the tuple.

Syntax: `tuple.count(element)`

Where the element is the element that is to be counted.

```
# Creating tuples
```

```
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
```

```
Tuple2 = ('python', 'plc', 'python', 'cad', 'java', 'python', 'IoT')
```

```
# count the appearance of 3
```

```
res = Tuple1.count(3)
```

```
print('Count of 3 in Tuple1 is:', res)
```

OUTPUT: Count of 3 in Tuple1 is: 3

```
# count the appearance of python
```

```
res1 = Tuple2.count('python')
```

```
print('Count of Python in Tuple2 is:', res1)
```

OUTPUT: Count of Python in Tuple2 is: 3

Index() Method

The Index() method returns the first occurrence of the given element from the tuple.

Syntax: `tuple.index(element, start, end)`

Parameters:

- **element:** The element to be searched.
- **start (Optional):** The starting index from where the searching is started
- **end (Optional):** The ending index till where the searching is done

Note: This method raises a ValueError if the element is not found in the tuple.

EXAMPLE:

Creating tuples

`Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)`

getting the index of 3

`res = Tuple.index(3)`

`print('First occurrence of 3 is', res)`

OUTPUT: First occurrence of 3 is 3

getting the index of 3 after 4th

index

`res = Tuple.index(3, 4)`

`print('First occurrence of 3 after 4th index is:', res)`

OUTPUT: First occurrence of 3 after 4th index is: 5

Advantages of Tuple over List in Python

- **Immutable:** Tuples are immutable, meaning that their contents cannot be modified once they are created. This can be useful in situations where you want to prevent accidental changes to the data, or when you need to ensure that the data remains constant throughout the lifetime of the program.
- **Faster:** Tuples are generally faster than lists for accessing and iterating over elements. This is because tuples are implemented as a contiguous block of memory, whereas lists are implemented as an array of pointers to objects.
- **Memory-efficient:** Tuples are more memory-efficient than lists, especially when the size of the collection is small. This is because tuples do not require additional space to store their length or other metadata, unlike lists.
- **Good for data integrity:** Tuples can be used to enforce data integrity, meaning that you can ensure that certain data is always present and in a specific format. For example, you might use a tuple to represent a coordinate in a two-dimensional space, where the first element is the x-coordinate and the second element is the y-coordinate.
- **Suitable for use as dictionary keys:** Tuples can be used as dictionary keys, whereas lists cannot. This is because tuples are immutable, and therefore their contents cannot change, which makes them suitable for use as a key in a hash table.

Python Set Data Type

- Sets are used to store multiple items in a single variable
- A set is a collection which is unordered (Unordered means that the items in a set do not have a defined order.), unchangeable, and unindexed.
- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Sets cannot have two items with the same value
- Sets are written with curly brackets.

Create a Set in Python

- A set can have any number of items and they may be of different types

Examples:

```
student_id = {112, 114, 116, 118, 115} # create a set of integer type  
print('Student ID:', student_id)
```

Output: Student ID: {112, 114, 115, 116, 118}

```
vowel_letters = {'a', 'e', 'i', 'o', 'u'} # create a set of string type  
print('Vowel Letters:', vowel_letters)
```

Output: Vowel Letters: {'u', 'a', 'e', 'i', 'o'}

```
mixed_set = {'Hello', 101, -2, 'Bye'} # create a set of mixed data types  
print('Set of mixed data types:', mixed_set)
```

Output: Set of mixed data types: {'Hello', 'Bye', 101, -2}

Create an Empty Set in Python

- Creating an empty set is a bit tricky. Empty curly braces { } will make an empty dictionary in Python.
- To make a set without any elements, we use the set() function without any argument

```
empty_set = set() # create an empty set
```

```
print('Data type of empty_set:', type(empty_set)) # check data type of empty_set
```

Output: Data type of empty_set: **<class 'set'>**

```
empty_dictionary = { } # create an empty dictionary
```

```
print('Data type of empty_dictionary', type(empty_dictionary)) # check data type of dictionary_set
```

Output: Data type of empty_dictionary **<class 'dict'>**

Duplicate Items in a Set

```
numbers = {2, 4, 6, 6, 2, 8}
```

```
print(numbers)
```

Output: {8, 2, 4, 6}

Python Set Methods

Functions Name	Description
<code>add()</code>	Adds a given element to a set
<code>clear()</code>	Removes all elements from the set
<code>copy()</code>	Returns a shallow copy of the set
<code>difference()</code>	Returns a set that is the difference between two sets
<code>difference_update()</code>	Updates the existing caller set with the difference between two sets
<code>discard()</code>	Removes the element from the set
<code>frozenset()</code>	Return an immutable frozenset object
<code>intersection()</code>	Returns a set that has the intersection of all sets
<code>intersection_update()</code>	Updates the existing caller set with the intersection of sets

Python Set Methods

Functions Name	Description
<code>remove()</code>	Removes the element from the set
<code>symmetric_difference()</code>	Returns a set which is the symmetric difference between the two sets
<code>symmetric_difference_update()</code>	Updates the existing caller set with the symmetric difference of sets
<code>union()</code>	Returns a set that has the union of all sets
<code>update()</code>	Adds elements to the set
<code>isdisjoint()</code>	Checks whether the sets are disjoint or not
<code>issubset()</code>	Returns True if all elements of a set A are present in another set B
<code>issuperset()</code>	Returns True if all elements of a set A occupies set B
<code>pop()</code>	Returns and removes a random element from the set

Built-in Functions with Python Set

Function	Description
<code>all()</code>	Returns <code>True</code> if all elements of the set are true (or if the set is empty).
<code>any()</code>	Returns <code>True</code> if any element of the set is true. If the set is empty, returns <code>False</code> .
<code>enumerate()</code>	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
<code>len()</code>	Returns the length (the number of items) in the set.
<code>max()</code>	Returns the largest item in the set.
<code>min()</code>	Returns the smallest item in the set.
<code>sorted()</code>	Returns a new sorted list from elements in the set(does not sort the set itself).
<code>sum()</code>	Returns the sum of all elements in the set.

Python Dictionaries Data Type

- 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.
- Dictionaries are written with curly brackets, and have keys and values:

Create a Dictionary

```
# creating a dictionary
country_capitals = {
    "United States": "Washington D.C.",
    "Italy": "Rome",
    "England": "London"
}
```

```
print(country_capitals) # printing the dictionary
```

Output:

```
{'United States': 'Washington D.C.', 'Italy': 'Rome', 'England': 'London'}
```

Python Dictionary Methods

Function	Description
<code>pop()</code>	Remove the item with the specified key.
<code>update()</code>	Add or change dictionary items.
<code>clear()</code>	Remove all the items from the dictionary.
<code>keys()</code>	Returns all the dictionary's keys.
<code>values()</code>	Returns all the dictionary's values.
<code>get()</code>	Returns the value of the specified key.
<code>popitem()</code>	Returns the last inserted key and value as a tuple.
<code>copy()</code>	Returns a copy of the dictionary.

Python Boolean Data Type

- **Python boolean** type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False.
- Python **bool()** function allows you to evaluate the value of any expression and returns either True or False based on the expression.

```
# Declaring variables
```

```
a = 10
```

```
b = 20
```

```
# Comparing variables
```

```
print(a == b)
```

Output:

False

Python Operators

Python – Operators

- Operators are used to perform operations on variables and values.
- Python operators are the constructs which can manipulate the value of operands.
- These are symbols used for the purpose of logical, arithmetic and various other operations.

Types of Python Operators

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Python Arithmetic Operators

- Python arithmetic operators are used to perform mathematical operations on numerical values.
- These operations are Addition, Subtraction, Multiplication, Division, Modulus, Exponents and Floor Division.

Operator	Name	Example
+	Addition	$10 + 20 = 30$
-	Subtraction	$20 - 10 = 10$
*	Multiplication	$10 * 20 = 200$
/	Division	$20 / 10 = 2$
%	Modulus	$22 \% 10 = 2$
**	Exponent	$4^{**}2 = 16$
//	Floor Division	$9//2 = 4$

Python Comparison Operators



- Python comparison operators compare the values on either sides of them and decide the relation among them. They are also called **relational operators**.
- These operators are equal, not equal, greater than, less than, greater than or equal to and less than or equal to.

Operator	Name	Example
==	Equal	4 == 5 is not true.
!=	Not Equal	4 != 5 is true.
>	Greater Than	4 > 5 is not true.
<	Less Than	4 < 5 is true.
>=	Greater than or Equal to	4 >= 5 is not true.
<=	Less than or Equal to	4 <= 5 is true.

Python Assignment Operators

- Python assignment operators are used to assign values to variables.
- These operators include simple assignment operator, addition assign, subtraction assign, multiplication assign, division and assign operators etc.

Operator	Name	Example
=	Assignment Operator	a = 10
+=	Addition Assignment	a += 5 (Same as a = a + 5)
-=	Subtraction Assignment	a -= 5 (Same as a = a - 5)
*=	Multiplication Assignment	a *= 5 (Same as a = a * 5)
/=	Division Assignment	a /= 5 (Same as a = a / 5)
%=	Remainder Assignment	a %= 5 (Same as a = a % 5)
**=	Exponent Assignment	a **= 2 (Same as a = a ** 2)
//=	Floor Division Assignment	a //= 3 (Same as a = a // 3)

Python Bitwise Operators

- Bitwise operator works on bits and performs bit by bit operation.
- There are following Bitwise operators supported by Python language

Operator	Name	Example
&	Binary AND	Sets each bit to 1 if both bits are 1
	Binary OR	Sets each bit to 1 if one of two bits is 1
^	Binary XOR	Sets each bit to 1 if only one of two bits is 1
~	Binary Ones Complement	Inverts all the bits
<<	Binary Left Shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Binary Right Shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Python Logical Operators

- There are following logical operators supported by Python language.

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Python Membership Operators

- Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.
- There are two membership operators

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

- Identity operators compare the memory locations of two objects.
- There are two Identity operators

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

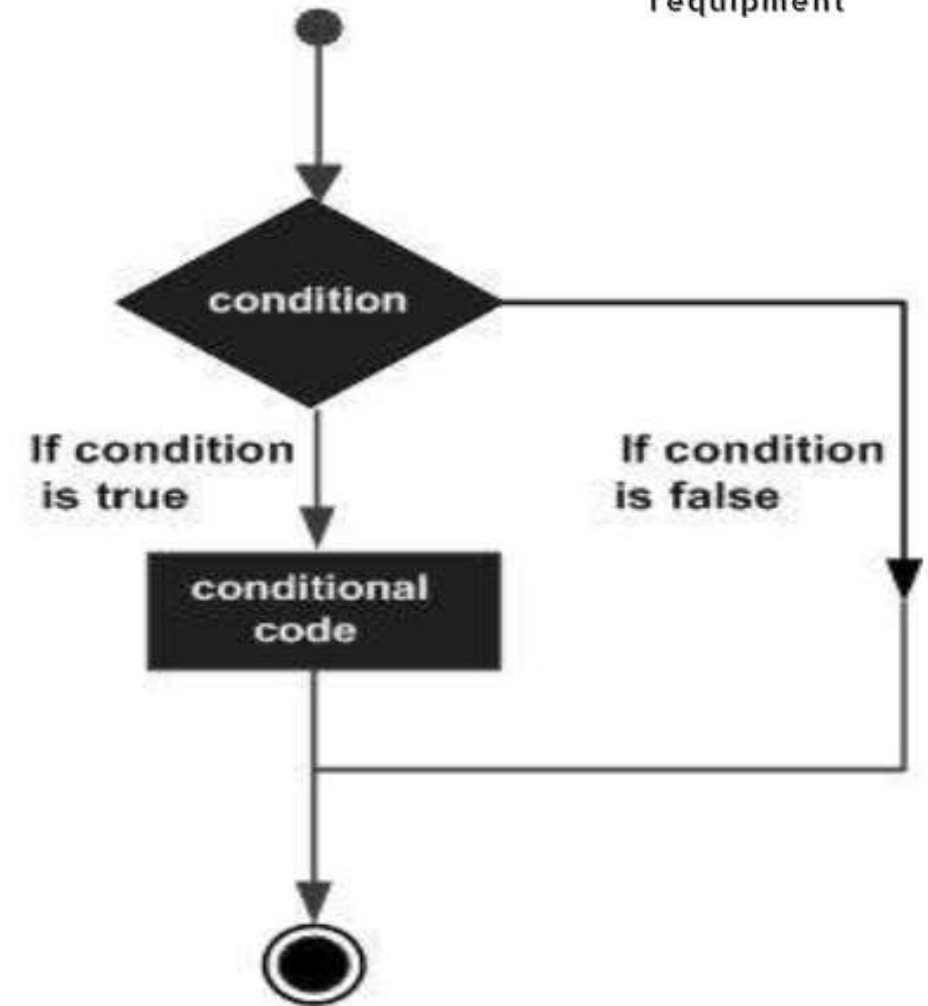
- The following table lists all operators from highest precedence to lowest.

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

Python - Decision Making

Python - Decision Making

- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
- Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome
- To determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.



- Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.
- Decision-making statements in programming languages decide the direction of the flow of program execution.
- In Python, if-else elif statement is used for decision making.

if statement

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

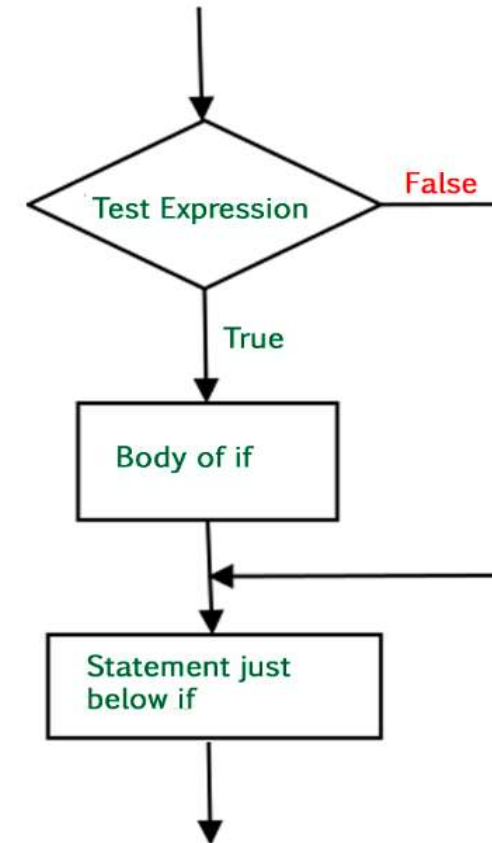
Syntax:

```
if condition:  
    statement 1  
    statement 2
```

- Python uses indentation to identify a block. So the block under an if statement will be identified

Flowchart of Python if statement

- Here, the condition after evaluation will be either true or false.
- if the statement accepts Boolean values – if the value is true then it will execute the block of statements below it otherwise not.



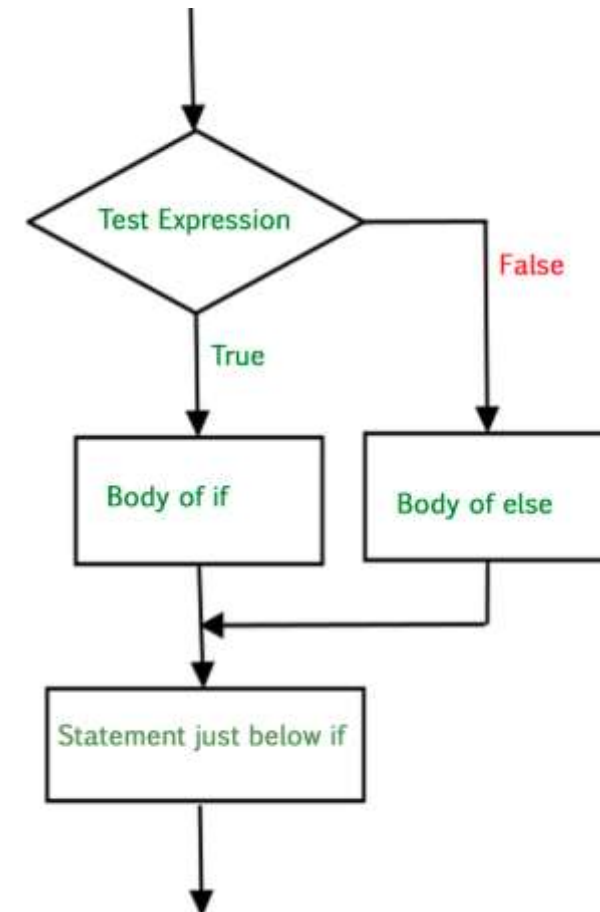
if-else:

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Flow Chart of Python If-else statement

Syntax:

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

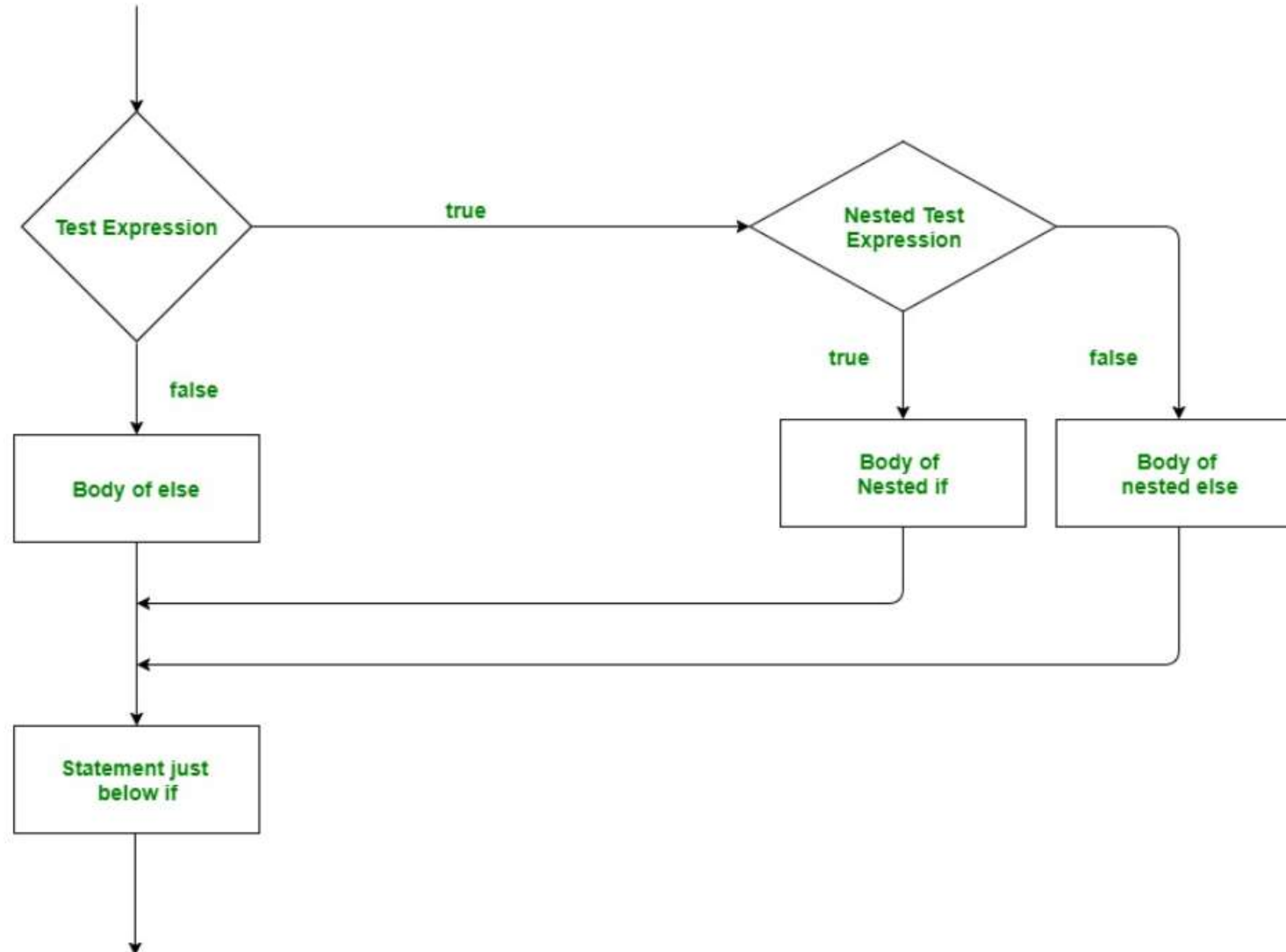


nested-if

- A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement.
- Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.
- **Syntax:**

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

Flowchart of Python Nested if Statement



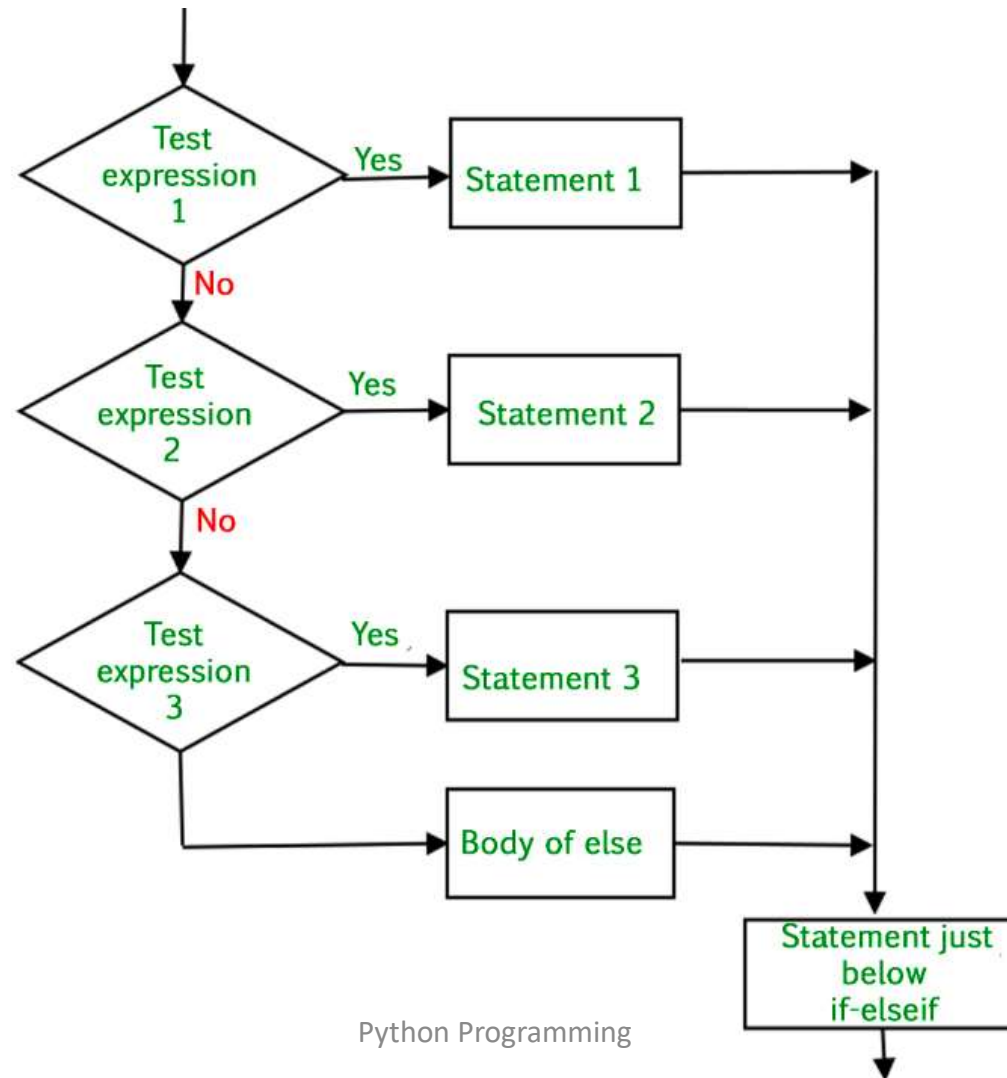
if-elif-else ladder



- Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.
- **Syntax:**

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

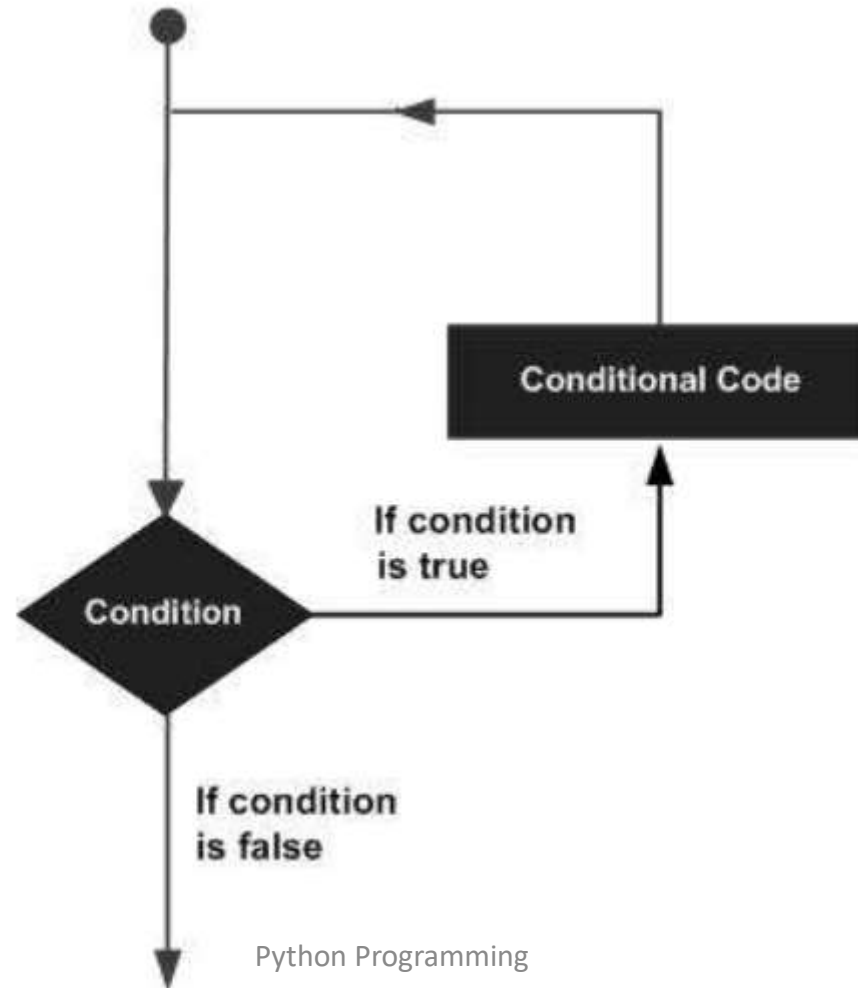

Flow Chart of Python if elif else statements



Python-Loops

Python – Loops

- A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement



- Python programming language provides following types of loops to handle looping requirements.
 1. **While loop:** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
 1. **For loop:** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
 2. **Nested loop:** You can use one or more loop inside any another while, for or do while loop.

Python While Loop

- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

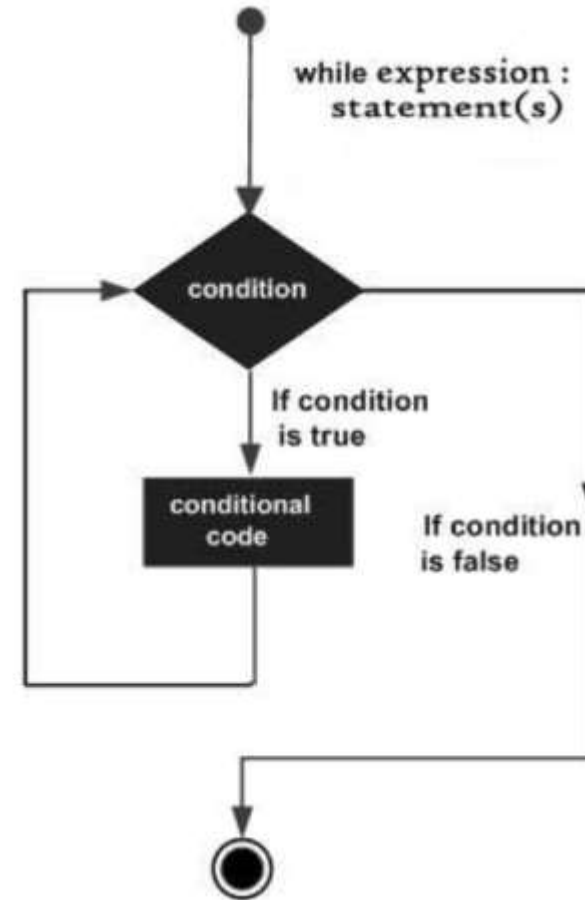
- **Syntax:**

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

- Here, **statement(s)** may be a single statement or a block of statements.
- The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- Python uses indentation as its method of grouping statements.

Python While Loop Flowchart

- In while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.



Python for Loop



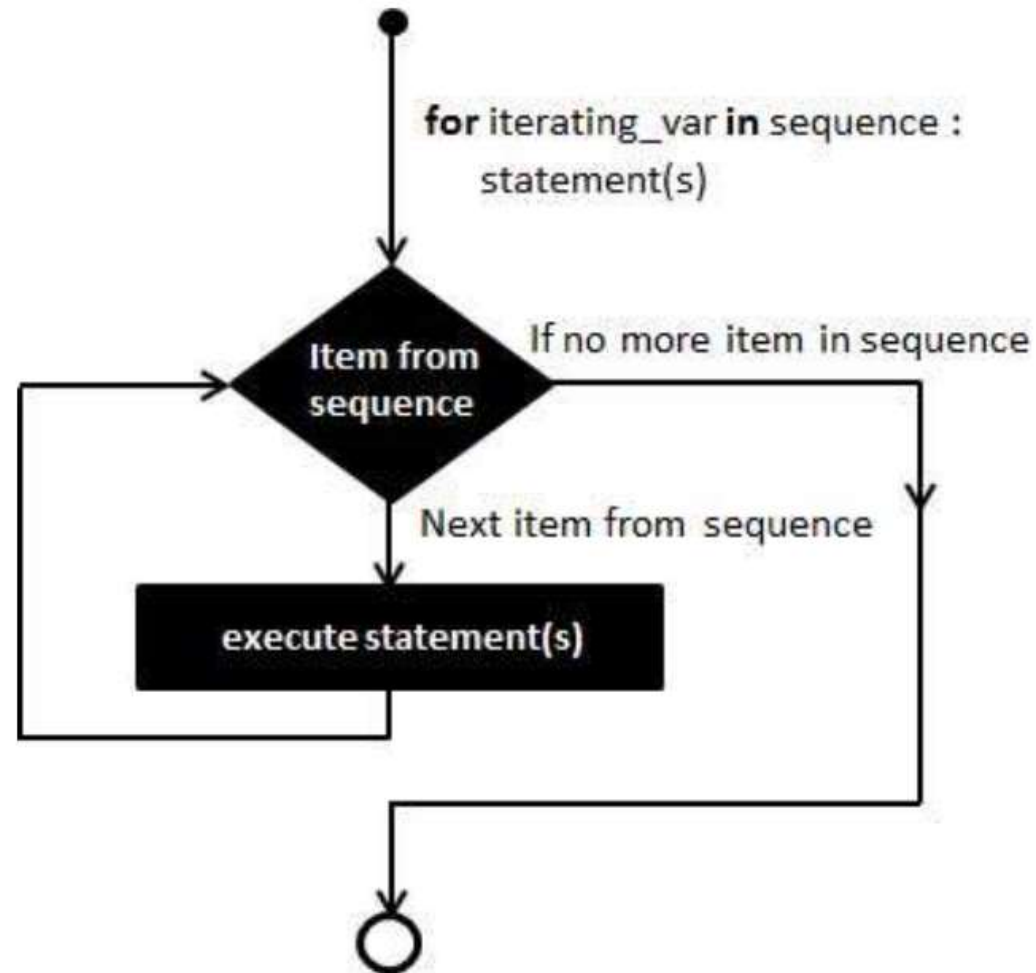
- It has the ability to iterate over the items of any sequence, such as a list or a string.

- **Syntax:**

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

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Python For Loop Flow Chart



Python nested loops



- Python programming language allows to use one loop inside another loop.

- **The syntax for nested for loop:**

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

- **The syntax for nested while loop:**

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

- In loop nesting is that you can put any type of loop inside of any other type of loop.
- For example a for loop can be inside a while loop or vice versa.

Loop Control Statements

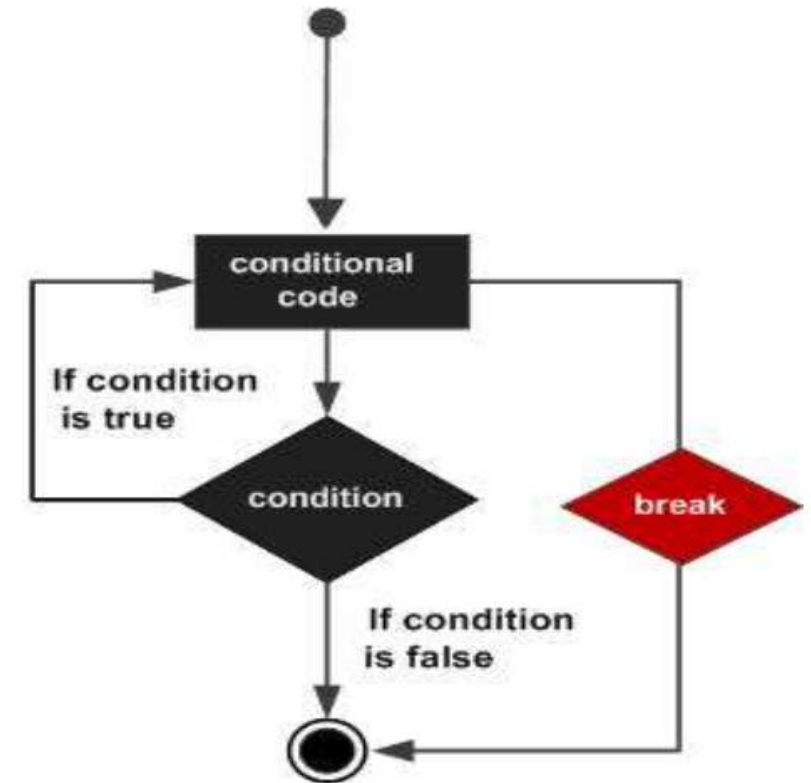


- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements.
 1. **Break statement** : Terminates the loop statement and transfers execution to the statement immediately following the loop
 2. **Continue statement** : Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
 3. **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.

Python break statement

- It terminates the current loop and resumes execution at the next statement.
- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop.
- The **break** statement can be used in both *while* and *for* loops.
- The syntax for a **break** statement in Python is: **break**

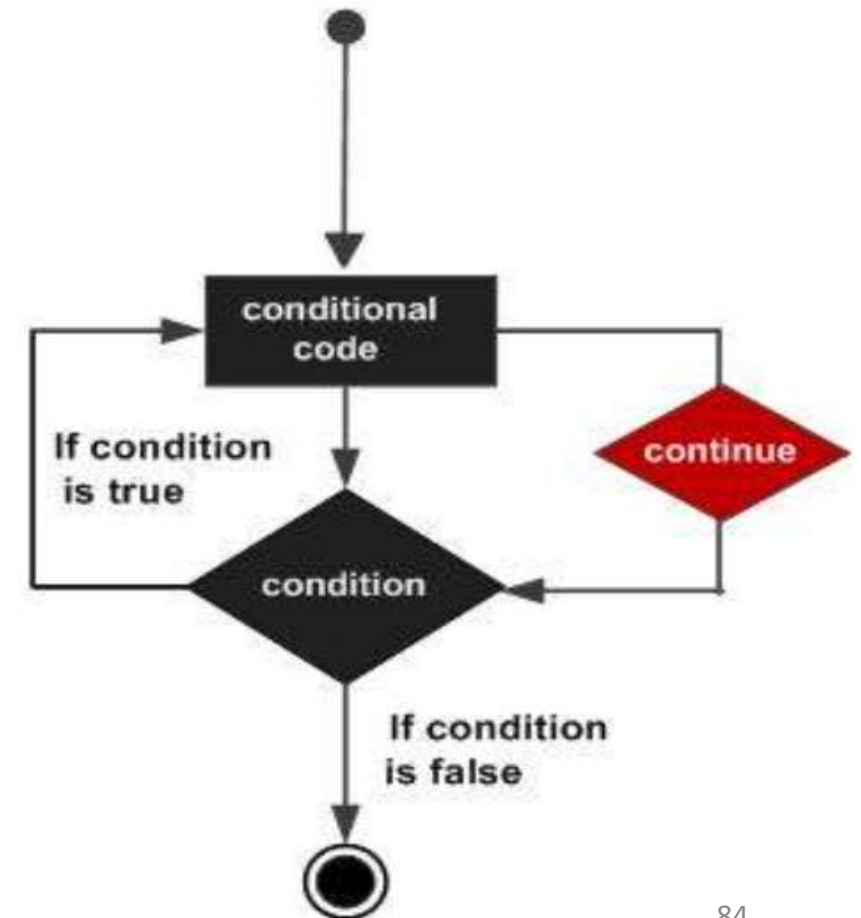
Flow Chart of Python break statement:



Python continue statement

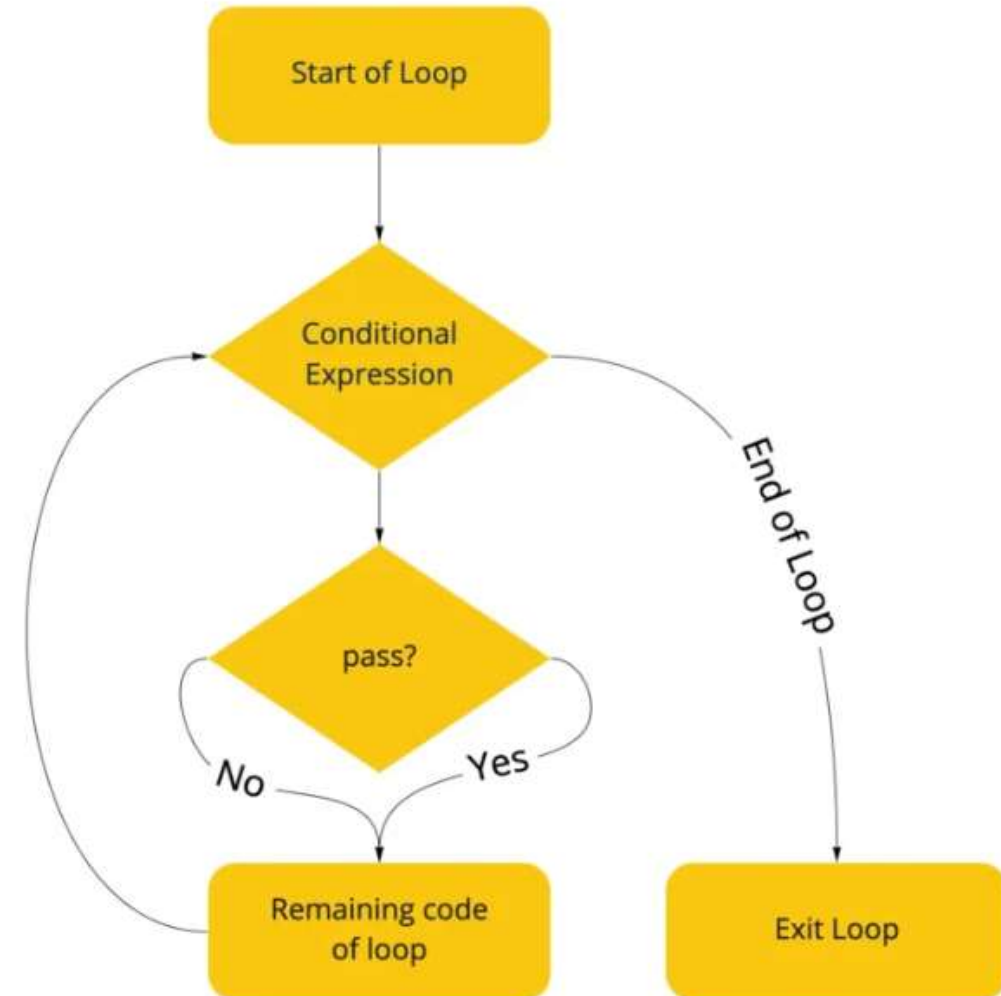
- It returns the control to the beginning of the while loop..
- The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- The **continue** statement can be used in both *while* and *for* loops.
- Syntax: **continue**

Flow Chart of Python continue statement:



Python pass Statement

- It is used when a statement is required syntactically but you do not want any command or code to execute.
- The **pass** statement is a *null* operation; nothing happens when it executes.
- The difference between pass and comment is that comment is ignored by the interpreter whereas pass is not ignored.
- The **pass** is also useful in places where your code will eventually go, but has not been written yet
- Syntax: **pass**



Python Function

Python Functions

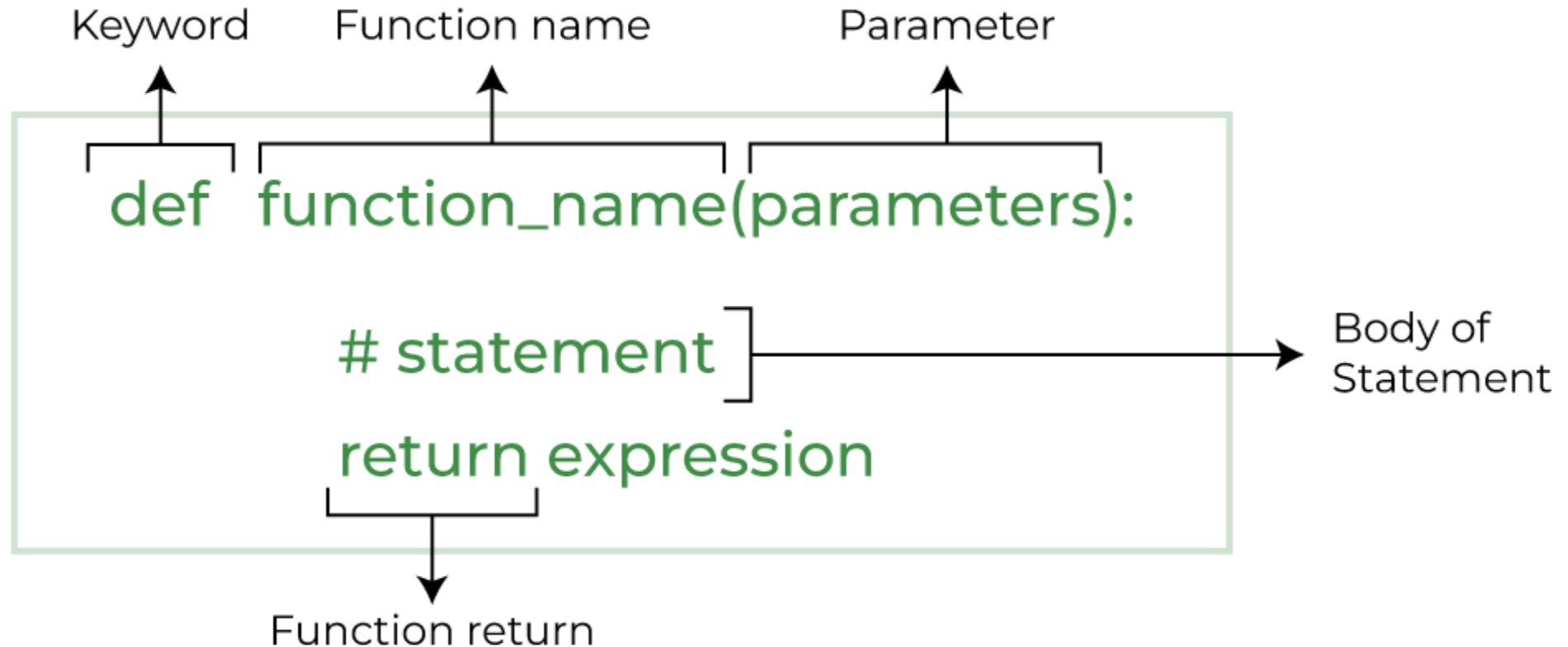
- A function is a block of code that performs a specific task.
- Function helps Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Benefits of Using Functions

1. **Code Reusable** - We can use the same function multiple times in our program which makes our code reusable.
2. **Code Readability** - Functions help us break our code into chunks to make our program readable and easy to understand.

Python Function Declaration

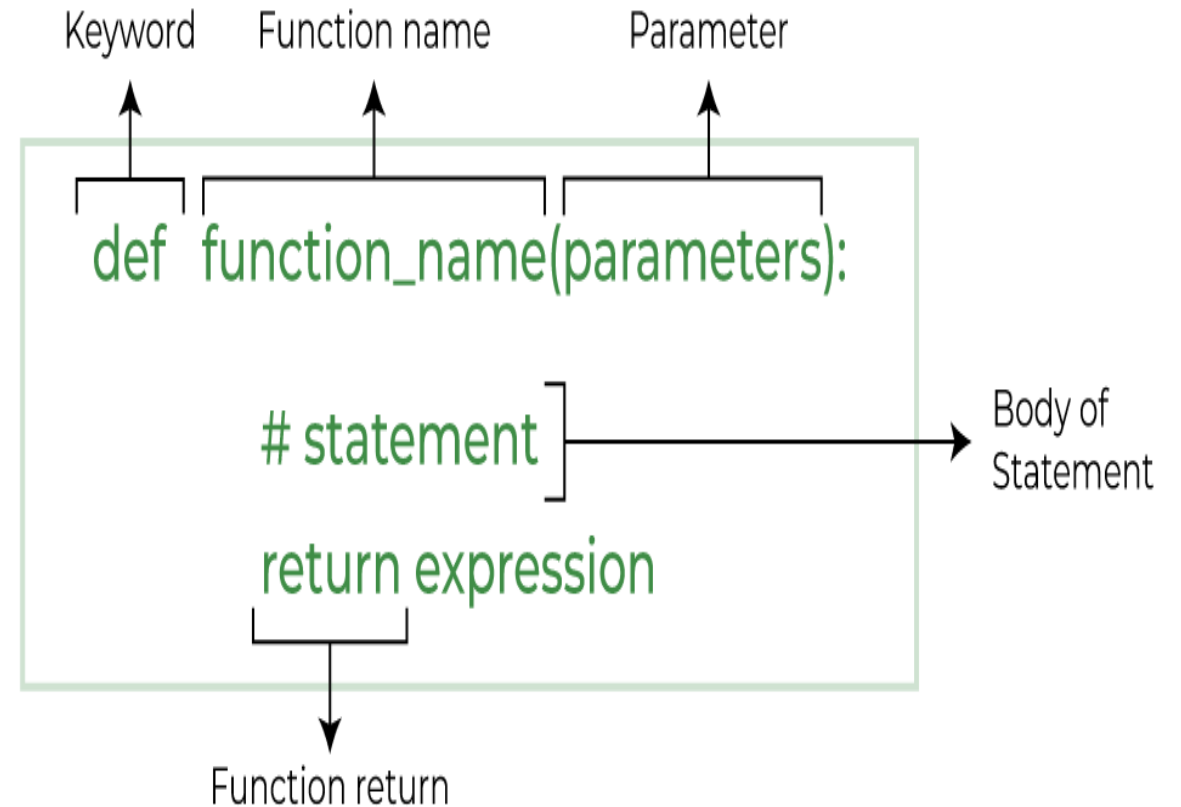
The syntax to declare a function is:



Python Function Declaration

The syntax to declare a function is:

- **def** - keyword used to declare a function
- **function_name** - any name given to the function
- **parameters** - any value passed to function
- **return** - (optional) - returns value from a function



Types of function

There are two types of function in Python programming:

- **Standard library functions** - These are built-in functions in Python that are available to use.

Examples: `print()`, `abs()`, `filter()`, `set()`, `map()`, `len()`, `pow()` etc.

- **User-defined functions** - We can create our own functions based on our requirements.

Creating a Function in Python

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

A simple python function

```
def fun( ):
```

```
    print("Well Come To GTTC")
```

Calling a Python Function

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

A simple python function

```
def fun( ):
```

```
    print("Well Come To GTTC")
```

Driver code to call a function

```
fun( )
```

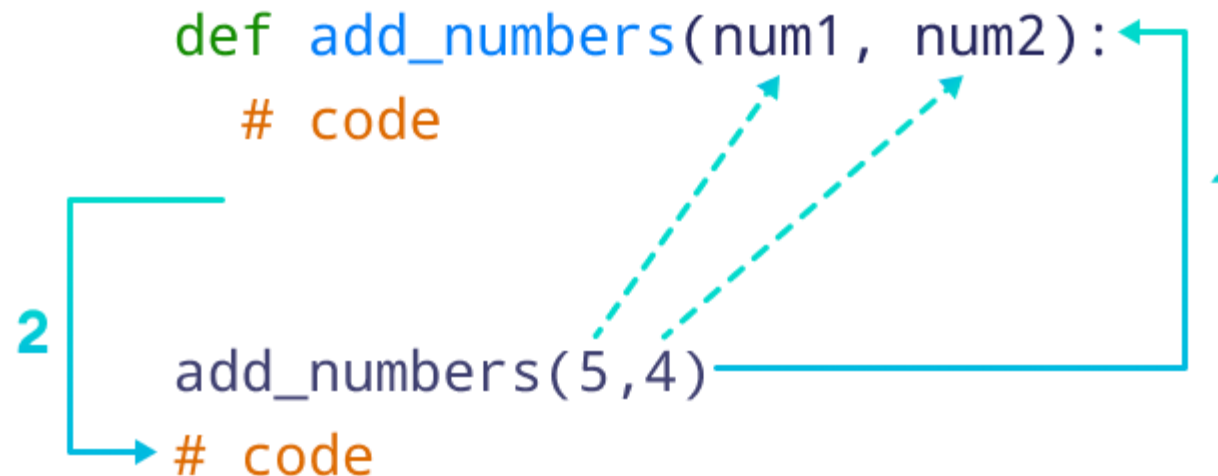
OUTPUT:

Well Come To GTTC

- When the function is called, the control of the program goes to the function definition.
- All codes inside the function are executed.
- The control of the program jumps to the next statement after the function call.

Python Function Arguments

- A function can also have arguments. An argument is a value that is accepted by a function.
- In this example the function is `add_numbers` with arguments are `num1, num2`



Types of Python Function Arguments



- Python supports various types of arguments that can be passed at the time of the function call.
- In Python, we have the following 4 types of function arguments.
 - **Default argument:** A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.
 - **Keyword arguments (named arguments):** The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.
 - **Positional arguments**
 - **Arbitrary arguments** (variable-length arguments `*args` and `**kwargs`)

Python Function Arguments Example



```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum: ",sum)
```

```
# function call with two values
add_numbers(5, 4)
```

```
# Output: Sum: 9
```


The return Statement in Python

- A Python function may or may not return a value. If we want our function to return some value to a function call, we use the return statement.

function with return type

```
def find_square(num):
```

```
    result = num * num
```

```
    return result
```

function call

```
square = find_square(3)
```

```
print(' Square:', square)
```

Output: Square: 9

Function Argument with Default Values



```
def add_numbers( a = 7, b = 8):  
    sum = a + b  
    print('Sum:', sum)
```

```
# function call with two arguments  
add_numbers(2, 3)
```

```
# function call with one argument  
add_numbers(a = 2)
```

```
# function call with no arguments  
add_numbers()
```

1. add_number(2, 3)

Both values are passed during the function call. Hence, these values are used instead of the default values.

Output: Sum: 5

2. add_number(2)

Only one value is passed during the function call. So, according to the positional argument **2** is assigned to argument **a**, and the default value is used for parameter **b**

Output: Sum: 10

3. add_number()

No value is passed during the function call. Hence, default value is used for both parameters **a** and **b**

Output: Sum: 15

Python Keyword Argument

- In keyword arguments, arguments are assigned based on the name of arguments.
- In keyword arguments the position of arguments doesn't matter

```
def display_info(first_name, last_name):  
    print('First Name:', first_name)  
    print('Last Name:', last_name)
```

```
display_info(last_name = 'Hubli', first_name = 'GTTC')
```

Output:

```
First Name: GTTC  
Last Name: Hubli
```

Python Function With Arbitrary Arguments



- Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments in Python.
- Arbitrary arguments allow us to pass a varying number of values during a function call.
- An asterisk (*) before the parameter name to denote this kind of argument.

program to find sum of multiple numbers

```
def find_sum(*numbers):
```

```
    result = 0
```

```
    for num in numbers:
```

```
        result = result + num
```

```
    print("Sum = ", result)
```

function call with 3 arguments

```
find_sum(1, 2, 3)
```

function call with 2 arguments

```
find_sum(4, 9)
```

- In the example, we have created the function `find_sum()` that accepts arbitrary arguments.
- Here, we are able to call the same function with different arguments.
- After getting multiple values, numbers behave as an array so we are able to use the for loop to access each value.

```
find_sum(1, 2, 3)
```

Output: Sum = 5

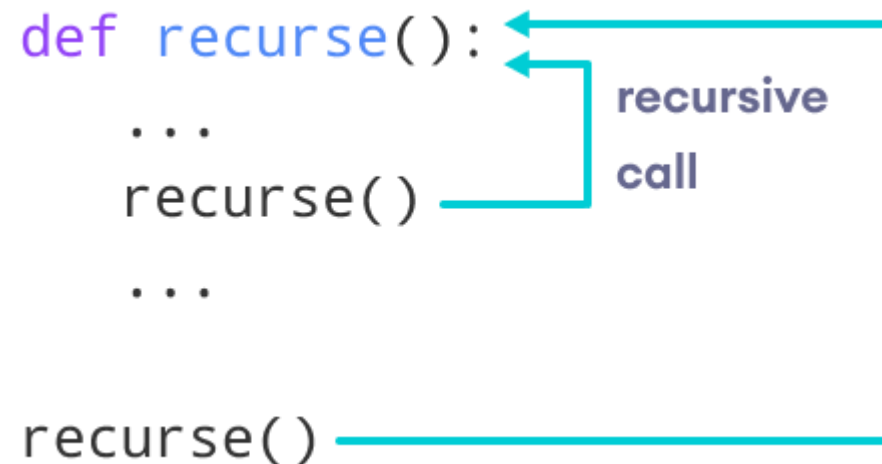
```
find_sum(4, 9)
```

Output: Sum = 13

Python Recursion

- Recursion is the process of defining something in terms of itself.
- In Python, the function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

```
def recurse():  
    ...  
    recurse()  
    ...  
  
recurse()
```



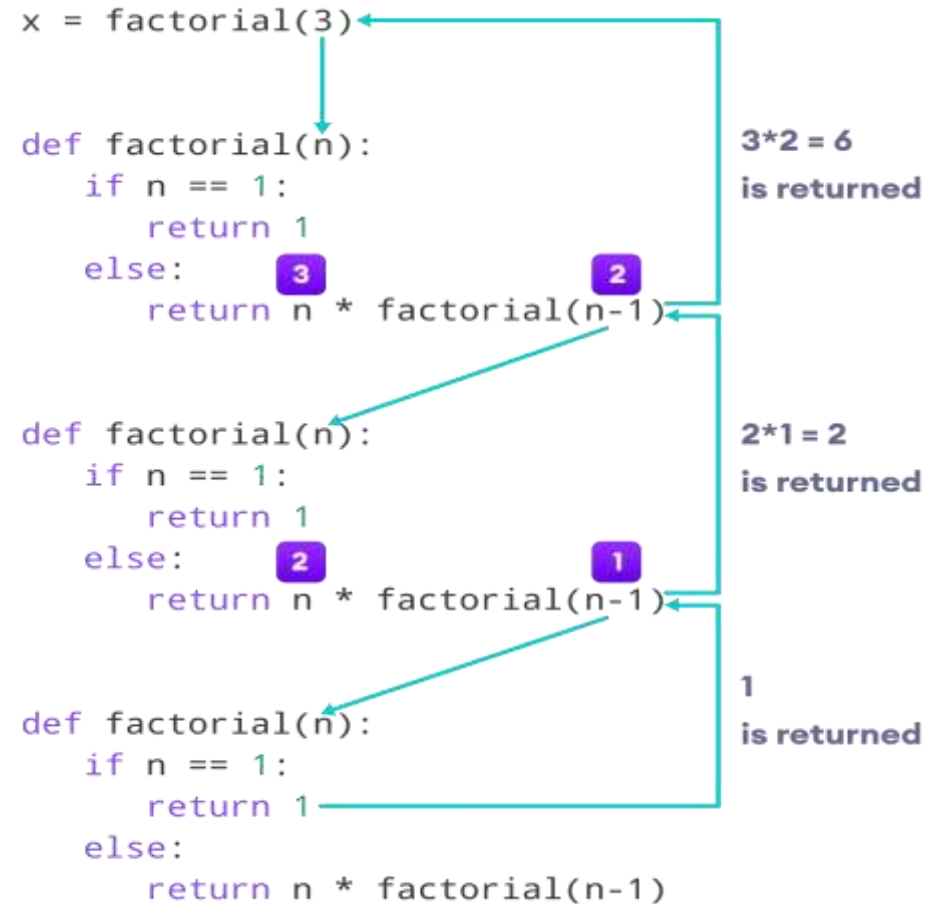
Example of a recursive function

This is a recursive function
to find the factorial of an integer

```
def factorial(x):  
    if x == 1:  
        return 1  
    else:  
        return (x * factorial(x-1))
```

```
num = 3  
print("The factorial of", num, "is",  
      factorial(num))
```

Output: The factorial of 3 is 6



Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Python Lambda/Anonymous Function



- In Python, a lambda function is a special type of function without the function name.
- The **lambda** keyword is used instead of **def** to create a lambda function.
- The syntax to declare the lambda function is
lambda argument(s) : expression

Here,

argument(s) - any value passed to the lambda function

expression - expression is executed and returned

Python lambda Function

declare a lambda function

```
course = lambda : print('Internet of Things')
```

call lambda function

```
course()
```

Output: Internet of Things

- In this example, we have defined a lambda function and assigned it to the `course` variable.
- When we call the **lambda** function, the `print()` statement inside the lambda function is executed.

Python lambda Function with an Argument

- Similar to normal functions, the lambda function can also accept arguments.

lambda that accepts one argument

```
course_name = lambda name : print('The course is,', name)
```

lambda call

```
course_name('Internet of Things')
```

```
course_name('Programmable Logic Controller')
```

Output: The course is, Internet of Things

Output: The course is, Programmable Logic Controller

Python lambda function with filter()

- The filter() function in Python takes in a function and an iterable (lists, tuples, and strings) as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Program to filter out only the even items from a list

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
```

```
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
```

```
print(new_list)
```

Output: [4, 6, 8, 12]

- Here, the filter() function returns only even numbers from a list

Python lambda function with map()

- The map() function in Python takes in a function and an iterable (lists, tuples, and strings) as arguments.
- The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Program to double each item in a list using map()

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
```

```
new_list = list(map(lambda x: x * 2 , my_list))
```

```
print(new_list)
```

Output: [2, 10, 8, 12, 16, 22, 6, 24]

- Here, the map() function doubles all the items in a list.

Functions Assignment

1. Print the Armstrong numbers in the given range using functions
2. Perform the calculator using functions
3. Print student grades using functions.
4. Perform the following based on the given choices
 - a. Greatest of two numbers
 - b. Equal to two numbers
5. Calculate the factorial of a given number
6. Print the Fibonacci series until the given range
7. Print the prime numbers in the given range
8. Print the reverse of the given number
9. Print the numbers which are divisible by 3 and 12 in the given range
10. Print the palindrome in the given range.

Python Files

Python File Operation

- A file is a container in computer storage devices used for storing data.
- When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.
- 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

Opening Files in Python

- In Python, we use the `open()` method to open files.
- To demonstrate how we open files in Python, let's suppose we have a file named **test.txt** with the following content.

```
test.txt
1  This is test file.
2  Hello from the test file.
3
4
5
6
7
```

- To open data from this file using the `open()` function.

```
# open file in current directory
```

```
file1 = open("test.txt")
```

Here, we have created a file object named file1. This object can be used to work with files and directories.

By default, the files are open in read mode (cannot be modified). The code above is equivalent to

```
file1 = open("test.txt", "r")
```

Here, we have explicitly specified the mode by passing the "r" argument which means file is opened for reading.

Different Modes to Open a File in Python



Mode	Description
r	Open a file for reading. (default)
w	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Open a file for exclusive creation. If the file already exists, the operation fails.
a	Open a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Open in text mode. (default)
b	Open in binary mode.
+	Open a file for updating (reading and writing)

to open a file in different modes,

```
file1 = open("test.txt")    # equivalent to 'r' or 'rt'
```

```
file1 = open("test.txt",'w') # write in text mode
```

```
file1 = open("img.bmp",'r+b') # read and write in binary mode
```

Reading Files in Python

After we open a file, we use the `read()` method to read its contents. For example,

```
# open a file
file1 = open("test.txt", "r")
# read the file
read_content = file1.read()
print(read_content)
```

Output

```
This is a test file.
Hello from the test file.
```

In the above example, we have read the `test.txt` file that is available in our current directory. Notice the code,

```
read_content = file1.read
```

Here, `file1.read()` reads the `test.txt` file and is stored in the `read_content` variable.

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the `close()` method in Python. For example,

```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)

# close the file
file1.close()
```

Output

```
This is a test file.
Hello from the test file.
```

Here, we have used the `close()` method to close the file.

After we perform file operation, we should always close the file; it's a good programming practice.

Exception Handling in Files

If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try...finally block.

Let's see an example,

try:

```
file1 = open("test.txt", "r")  
read_content = file1.read()  
print(read_content)
```

finally:

```
# close the file  
file1.close()
```

Here, we have closed the file in the finally block as finally always executes, and the file will be closed even if an exception occurs.

Use of with...open Syntax

In Python, we can use the with...open syntax to automatically close the file.

For example,

```
with open("test.txt", "r") as file1:  
    read_content = file1.read()  
    print(read_content)
```

Here we don't have to worry about closing the file, make a habit of using the with...open syntax.

Writing to Files in Python

There are two things we need to remember while writing to a file.

- If we try to open a file that doesn't exist, a new file is created.
- If a file already exists, its content is erased, and new content is added to the file.

In order to write into a file in Python, we need to open it in write mode by passing "w" inside open() as a second argument.

Suppose, we don't have a file named test2.txt. Let's see what happens if we write contents to the **test2.txt file**.

with open(test2.txt', 'w') as file2:

```
# write contents to the test2.txt file  
file2.write('Internet of Things.')  
fil2.write('Govt. Tool Room Training Center')
```

Here, a new test2.txt file is created and this file will have contents specified inside the write() method.

Python File Methods

There are various methods available with the file object

Method	Description
<code>close()</code>	Closes an opened file. It has no effect if the file is already closed.
<code>detach()</code>	Separates the underlying binary buffer from the <code>TextIOBase</code> and returns it.
<code>fileno()</code>	Returns an integer number (file descriptor) of the file.
<code>flush()</code>	Flushes the write buffer of the file stream.
<code>isatty()</code>	Returns <code>True</code> if the file stream is interactive.

<code>read(n)</code>	Reads at most n characters from the file. Reads till end of file if it is negative or <code>None</code> .
<code>readable()</code>	Returns <code>True</code> if the file stream can be read from.
<code>readline(n=-1)</code>	Reads and returns one line from the file. Reads in at most n bytes if specified.
<code>readlines(n=-1)</code>	Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified.
<code>seek(offset,from=SEEK_SET)</code>	Changes the file position to offset bytes, in reference to from (start, current, end).
<code>seekable()</code>	Returns <code>True</code> if the file stream supports random access.
<code>tell()</code>	Returns an integer that represents the current position of the file's object.
<code>truncate(size=None)</code> current location.	Resizes the file stream to size bytes. If size is not specified, resizes to
<code>writable()</code>	Returns <code>True</code> if the file stream can be written to.
<code>write(s)</code>	Writes the string s to the file and returns the number of characters written.
<code>writelines(lines)</code>	Writes a list of lines to the file.

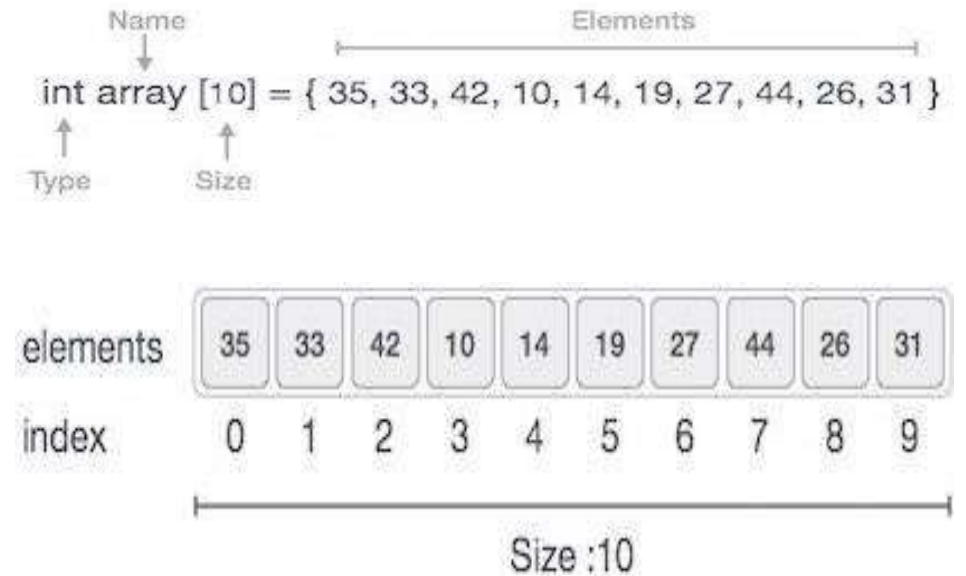
PYTHON ARRAY

Python Array

- The Array is a process of memory allocation. It is performed as a dynamic memory allocation.
- It is a container that can hold a fixed number of items, and these items should be the same type.
- The Array is an idea of storing multiple items of the same type together, making it easier to calculate the position of each element by simply adding an offset to the base value.
- A combination of the arrays could save a lot of time by reducing the overall size of the code.
- It is used to store multiple values in a single variable.

Array Representation

- Arrays can be declared in various ways in different languages
- As per the diagram Index starts with 0.
 - Array length is 10, which means it can store 10 elements.
 - Each element can be accessed via its index. For example, we can fetch an element at index 6 as 27.



Basic Operations Of Array

The basic operations supported by an array are

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

Creating an Array

- The array can be handled in Python by a module named array.
- Array in Python can be created by importing an array module.

from array import *

arrayName = array(typecode, [Initializers])

- Here, **typecode** are the codes that are used to define the type of value the array will hold

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Example

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
for x in array1:  
    print(x)  
Here array1 is the name of the array
```

Output:

10
20
30
40
50

Accessing Array Element

- We can access each element of an array using the index of the element.

Example:

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
print (array1[0])  
print (array1[2])
```

Output

10

30

Insertion Operation

- Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Example:

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
array1.insert(1,60)  
for x in array1:  
    print(x)
```

Output:

- It produces the following result which shows the element is inserted at index position 1.

```
10  
60  
20  
30  
40  
50
```

Deletion Operation

- Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Example

- Here, we remove a data element at the middle of the array using the python in-built `remove()` method.

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
array1.remove(40)  
for x in array1:  
    print(x)
```

Output

- It produces the following result which shows the element is removed from the array.

```
10  
20  
30  
50
```

Search Operation

- Search for an array element based on its value or its index.

Example

Here, we search a data element using the python in-built `index()` method.

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
print (array1.index(40))
```

Output

- It produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.

3

Update Operation

- Update operation refers to updating an existing element from the array at a given index. Example
- Here, we simply reassign a new value to the desired index we want to update.

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
array1[2] = 80  
for x in array1:  
    print(x)
```

Output

- When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

```
10  
20  
80  
40  
50
```

Python Classes and Objects

- A class is a user-defined blueprint or prototype from which objects are created.
- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state.
- Class instances can also have methods (defined by their class) for modifying their state.
- The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Python Classes

- The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
Eg.: My class.Myattribute

Syntax: Class Definition

```
class ClassName:  
    # Statement
```

Creating a Python Class

- Here, the class keyword indicates that you are creating a class followed by the name of the class

Example:

```
class Dog  
    sound = "bark"
```

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        # This is the constructor method that is called when creating a new Person object
```

```
        # It takes two parameters, name and age, and initializes them as attributes of the object
```

```
        self.name = name
```

```
        self.age = age
```

```
    def greet(self):
```

```
        # This is a method of the Person class that prints a greeting message
```

```
        print("Hello, my name is " + self.name)
```

- **Name and age are the two properties of the Person class. Additionally, it has a function called greet that prints a greeting.**

Python Objects

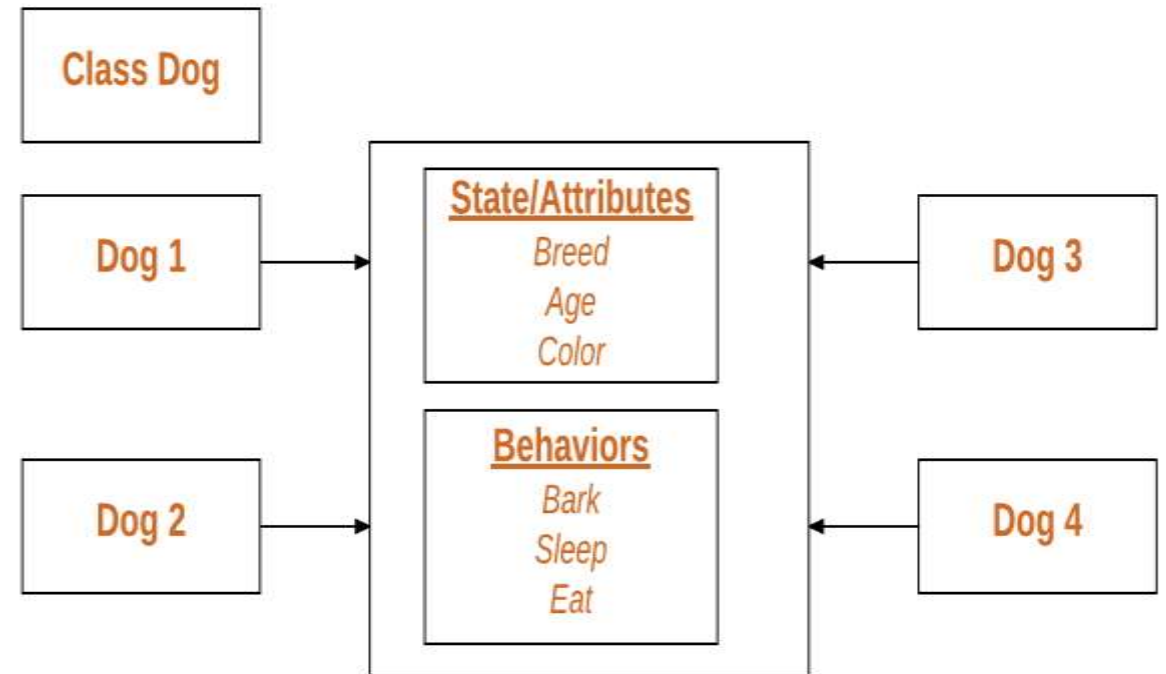
- An Object is an instance of a Class.
- A class is like a blueprint while an instance is a copy of the class with actual values
- An object consists of:
 - **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
 - **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
 - **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example:



Declaring Class Objects

- When an object of a class is created, the class is said to be instantiated.
- All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object.
- A single class may have any number of instances



- An object is a particular instance of a class with unique characteristics and functions.
- After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python.
- The object's attributes are initialised in the constructor, which is a special procedure with the name `__init__`.

Syntax:

```
# Declare an object of a class  
object_name = Class_Name(arguments)
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        print("Hello, my name is " + self.name)
```

```
# Create a new instance of the Person class and assign it to the variable person1
person1 = Person("Ayan", 25)
person1.greet()
```

Output:

"Hello, my name is Ayan"

Python Inheritance

- Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.

- Inheritance allows us to create a new class from an existing class.
- The new class that is created is known as subclass (child or derived class) and the existing class from which the child class is derived is known as superclass (parent or base class).

Python Inheritance Syntax:

```
# define a superclass
class super_class:
    # attributes and method definition
# inheritance
class sub_class(super_class):
    # attributes and method of super_class
    # attributes and method of sub_class
```

Here, we are inheriting the sub_class class from the super_class class.

Example:

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

dog barking

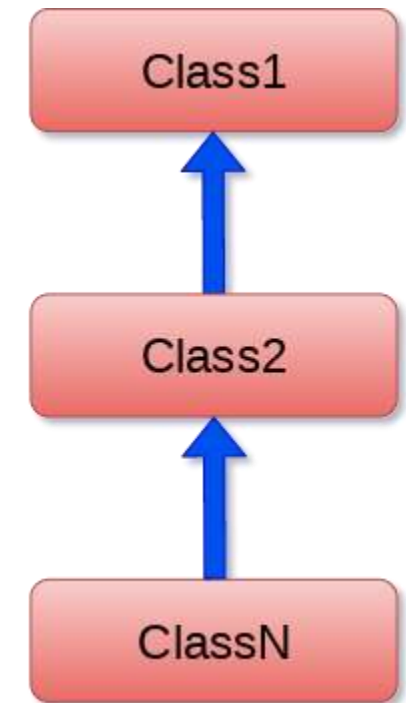
Animal Speaking

Python Multi-Level inheritance

- Multi-Level inheritance is possible in python like other object-oriented languages.
- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

The syntax of multi-level inheritance is

```
class class1:  
    <class-suite>  
class class2(class1):  
    <class suite>  
class class3(class2):  
    <class suite>
```



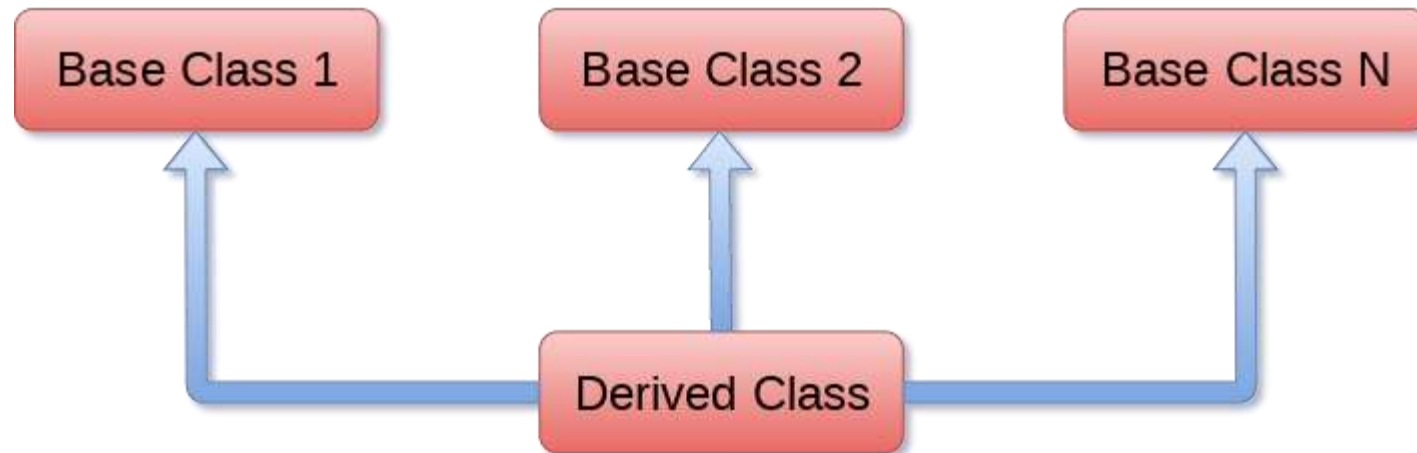
Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:
dog barking
Animal Speaking
Eating bread...

Python Multiple inheritance

- Python provides us the flexibility to inherit multiple base classes in the child class



The syntax to perform multiple inheritance is

```
class Base1:
```

```
    <class-suite>
```

```
class Base2:
```

```
    <class-suite>
```

```
·
```

```
·
```

```
·
```

```
class BaseN:
```

```
    <class-suite>
```

```
class Derived(Base1, Base2, ..... BaseN):
```

```
    <class-suite>
```

Example:

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

30

200

0.5

Polymorphism in Python

- Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

Function Polymorphism in Python

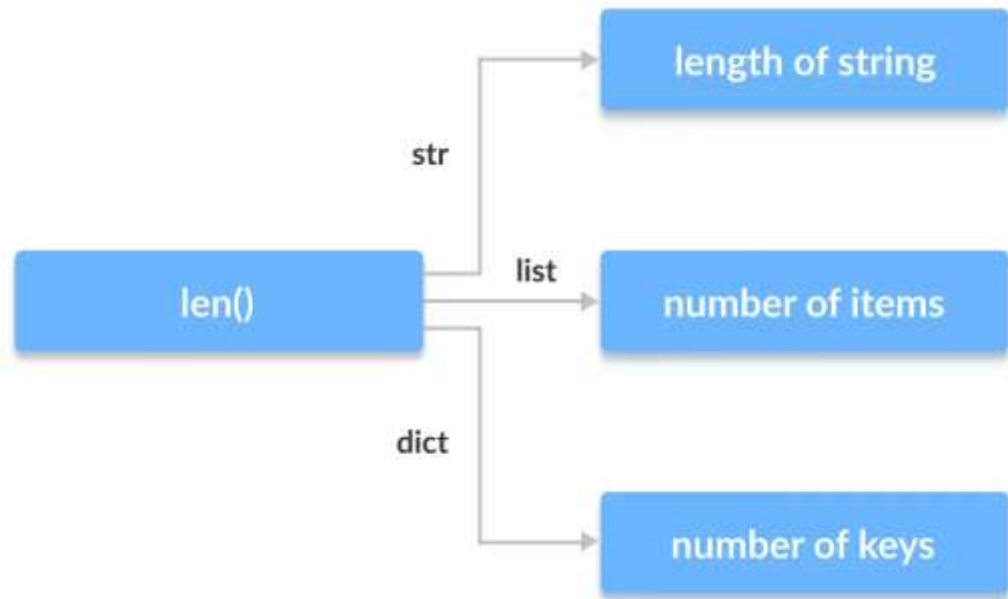
- There are some functions in Python which are compatible to run with multiple data types.
- One such function is the len() function. It can run with many data types in Python. Let's look at some example use cases of the function.

Example: Polymorphic len() function:

```
print(len("GTTC"))  
print(len(["PLC", "IoT", "Automation"]))  
print(len({"Name": "Asha", "Address": "Hubli"}))
```

Output:

```
4  
3  
2
```



- Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the `len()` function.
- However, we can see that it returns specific information about specific data types.

Python Exceptions

- When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception.
- An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.
- When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.

For instance, they occur when we

- try to open a file(for reading) that does not exist (FileNotFoundError)
- try to divide a number by zero (ZeroDivisionError)
- try to import a module that does not exist (ImportError) and so on.

Whenever these types of runtime errors occur, Python creates an exception object.

If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Some of the common built-in exceptions in Python programming along with the error that cause them are

Exception	Cause of Error
<code>AssertionError</code>	Raised when an <code>assert</code> statement fails.
<code>AttributeError</code>	Raised when attribute assignment or reference fails.
<code>EOFError</code>	Raised when the <code>input()</code> function hits end-of-file condition.
<code>FloatingPointError</code>	Raised when a floating point operation fails.
<code>GeneratorExit</code>	Raise when a generator's <code>close()</code> method is called.
<code>ImportError</code>	Raised when the imported module is not found.
<code>IndexError</code>	Raised when the index of a sequence is out of range.

<code>KeyError</code>	Raised when a key is not found in a dictionary.
<code>KeyboardInterrupt</code>	Raised when the user hits the interrupt key (<code>Ctrl+C</code> or <code>Delete</code>).
<code>MemoryError</code>	Raised when an operation runs out of memory.
<code>NameError</code>	Raised when a variable is not found in local or global scope.
<code>NotImplementedError</code>	Raised by abstract methods.
<code>OSError</code>	Raised when system operation causes system related error.
<code>OverflowError</code>	Raised when the result of an arithmetic operation is too large to be represented.
<code>ReferenceError</code>	Raised when a weak reference proxy is used to access a garbage collected referent.
<code>RuntimeError</code>	Raised when an error does not fall under any other category.
<code>StopIteration</code>	Raised by <code>next()</code> function to indicate that there is no further item to be returned by iterator.

<code>SyntaxError</code>	Raised by parser when syntax error is encountered.
<code>IndentationError</code>	Raised when there is incorrect indentation.
<code>TabError</code>	Raised when indentation consists of inconsistent tabs and spaces.
<code>SystemError</code>	Raised when interpreter detects internal error.
<code>SystemExit</code>	Raised by <code>sys.exit()</code> function.
<code>TypeError</code>	Raised when a function or operation is applied to an object of incorrect type.
<code>UnboundLocalError</code>	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
<code>UnicodeError</code>	Raised when a Unicode-related encoding or decoding error occurs.

<code>UnicodeEncodeError</code>	Raised when a Unicode-related error occurs during encoding.
<code>UnicodeDecodeError</code>	Raised when a Unicode-related error occurs during decoding.
<code>UnicodeTranslateError</code>	Raised when a Unicode-related error occurs during translating.
<code>ValueError</code>	Raised when a function gets an argument of correct type but improper value.
<code>ZeroDivisionError</code>	Raised when the second operand of division or modulo operation is zero.

Python Error and Exception

- Errors represent conditions such as compilation error, syntax error, error in the logical part of the code, library incompatibility, infinite recursion, etc.
- Errors are usually beyond the control of the programmer and we should not try to handle errors.
- Exceptions can be caught and handled by the program.

Python Exception Handling

The exceptions abnormally terminate the execution of a program. In Python, we use the `try...except` block to handle exceptions.

The syntax of `try...except` block:

`try:`

 # code that may cause exception

`except:`

 # code to run when exception occurs

- Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by an except block.
- When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

Example: Exception Handling Using try...except

try:

```
numerator = 10
```

```
denominator = 0
```

```
result = numerator/denominator
```

```
print(result)
```

except:

```
print("Error: Denominator cannot be 0.")
```

Output:

Error: Denominator cannot be 0.

- In the example, we are trying to divide a number by 0. Here, this code generates an exception.
- To handle the exception, we have put the code, `result = numerator/denominator` inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped.
- The except block catches the exception and statements inside the except block are executed.
- If none of the statements in the try block generates an exception, the except block is skipped

Reference

- <https://www.w3schools.com/python/>
- <https://www.geeksforgeeks.org/python-programming-language/>
- <https://www.tutorialspoint.com/python/index.htm>
- <https://www.programiz.com/python-programming/online-compiler/>
- <https://www.codecademy.com/learn/learn-python>