Unit-I

Introduction to Python

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Python is the language of choice for **data analysis** and **machine learning**, but it can also adapt to create games and work with embedded devices

Technically speaking it is an interpreted language that does not have an intermediate compilation phase like a compiled language, for example C or Java.

Python supports a wide variety of different programming paradigms, including procedural programming, object oriented programming and functional programming.

Features of Python:

Python is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming.
In Python, we don't need to declare the type of variable because it is a dynamically typed language.
For example, x = 10
Here, x can be anything such as String, int, etc.

There are many features in Python, some of which are discussed below –

1. Easy to code:
Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in python language and anybody can learn python basics in a few hours or days. It is also a developer-friendly language.

2. Free and Open Source:
Python language is freely available at the official website and you can download it.
Since it is open-source, this means that source code is also available to the public. So you can download it as, use it as well as share it.

3. Object-Oriented Language:
One of the key features of python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, objects encapsulation, etc.

4. GUI Programming Support:
Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python.
PyQt5 is the most popular option for creating graphical apps with Python.

5. High-Level Language:
Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.

6. Extensible Language:
Python is a **Extensible** language. We can write us some Python code into C or C++ language and also we can compile that code in C/C++ language.

7. Python is Portable language:
Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

8. Python is Integrated language:
Python is also an Integrated language because we can easily integrated python with other languages like c, c++, etc.

9. Interpreted Language:
Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile python code this makes it easier to debug our code. The source code of python is converted into an immediate form called **bytecode**.

10. Large Standard Library
Python has a large standard library which provides a rich set of module and functions so you do not have to write your own code for every single thing. There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.

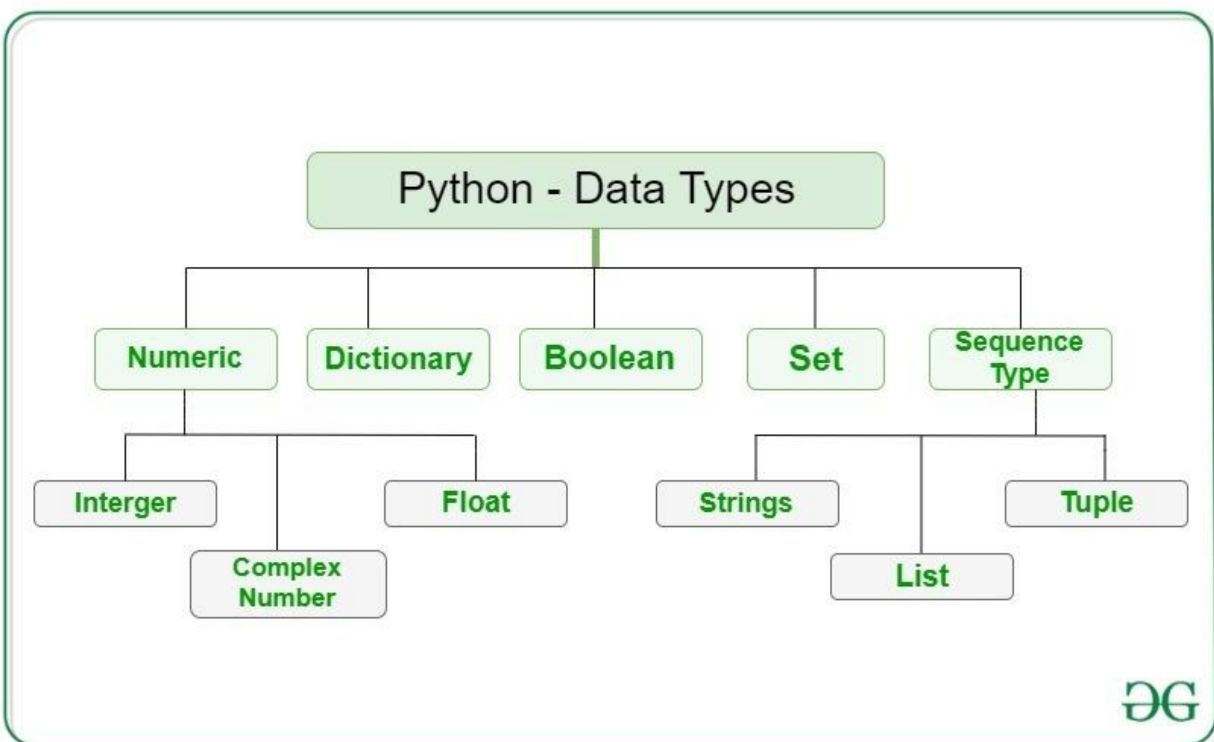
11. Dynamically Typed Language:
Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don’t need to specify the type of variable.

Data types:

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

Following are the standard or built-in data type of Python:

- **Numeric**
- **Sequence Type**
- **Boolean**
- **Set**
- **Dictionary**



Numeric:

In Python, numeric data type represent the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float

and complex class in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.

- **Float** – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.

- **Complex Numbers** – Complex number is represented by complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

Note – type() function is used to determine the type of data type.

```
# Python program to demonstrate numeric value

a = 5
print("Type of a: ", type(a))

b = 5.0
print("\nType of b: ", type(b))

c = 2 + 4j
print("\nType of c: ", type(c))
```

Output:

```
Type of a:  <class 'int'>

Type of b:  <class 'float'>

Type of c:  <class 'complex'>
```

Sequence Type:

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

- String

- List
- Tuple

String

In Python, Strings are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote or triple quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

Creating String

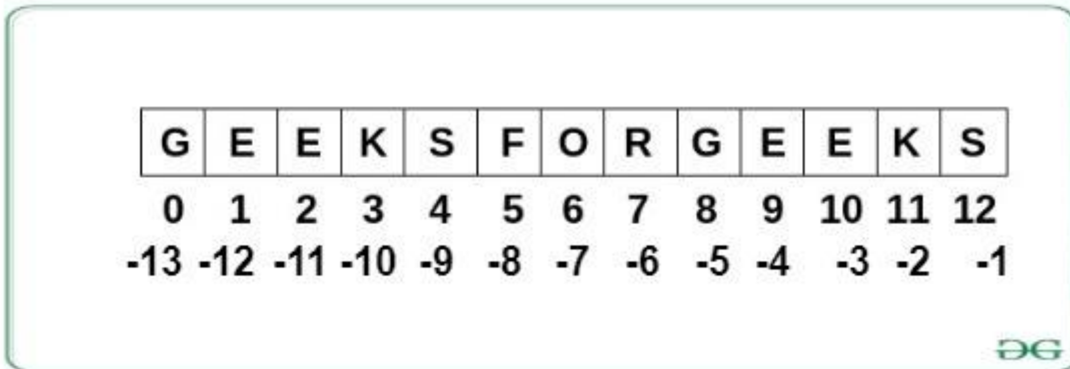
```
# Creating a String
# with single Quotes
String1 = 'Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)

# Creating a String
# with double Quotes
String1 = "I'm a Geek"
print("\nString with the use of Double Quotes: ")
print(String1)
print(type(String1))
```

Output:

```
String with the use of Single Quotes:
Welcome to the Geeks World

String with the use of Double Quotes:
I'm a Geek
<class 'str'>
```

Accessing elements of String

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character and so on.




```
String1 = "GeeksForGeeks"
print("Initial String: ")
print(String1)
# Printing First character
print("\nFirst character of String is: ")
print(String1[0])

# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])
```

Output:

```
Initial String:
GeeksForGeeks

First character of String is:
G

Last character of String is:
S
```

List

Lists are just like the arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

Creating List
Lists in Python can be created by just placing the sequence inside the square brackets[].

```
# Python program to demonstrate
```

```
# Creation of List
# Creating a List
List = []
print("Intial blank List: ")
print(List)

# Creating a List with
# the use of a String
List = ['GeeksForGeeks']
print("\nList with the use of String: ")
print(List)

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:

```
Intial blank List:
[]

List with the use of String:
['GeeksForGeeks']

List containing multiple values:
Geeks
Geeks

Multi-Dimensional List:
[['Geeks', 'For'], ['Geeks']]
```

Accessing elements of List

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. In Python, negative sequence indexes represent positions from the

end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
# Python program to demonstrate
# accessing of element from list
# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]
# accessing a element from the
# list using index number
print("Accessing element from the list")
print(List[0])
print(List[2])

# accessing a element using
# negative indexing
print("Accessing element using negative indexing")
# print the last element of list
print(List[-1])
# print the third last element of list
print(List[-3])
```

Output:

```
Accessing element from the list
Geeks
Geeks
Accessing element using negative indexing
Geeks
Geeks
```

Tuple

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

Creating Tuple

In Python, tuples are created by placing a sequence of values separated by ‘comma’ with or without the use of parentheses for grouping of the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.).

Note: Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing ‘comma’ to make it a tuple.

```
# Creating an empty tuple
Tuple1 = ()
print("Initial empty Tuple: ")
print (Tuple1)

# Creating a Tuple with
# the use of Strings
Tuple1 = ('Geeks', 'For')
print("\nTuple with the use of String: ")
print(Tuple1)

# Creating a Tuple with
# the use of list
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
print(tuple(list1))

# Creating a Tuple with the
# use of built-in function
Tuple1 = tuple('Geeks')
print("\nTuple with the use of function: ")
print(Tuple1)

# Creating a Tuple
# with nested tuples
Tuple1 = (0, 1, 2, 3)
Tuple2 = ('python', 'geek')
Tuple3 = (Tuple1, Tuple2)
print("\nTuple with nested tuples: ")
print(Tuple3)
```

Output:

```
Initial empty Tuple:
()

Tuple with the use of String:
('Geeks', 'For')

Tuple using List:
```

```
(1, 2, 4, 5, 6)

Tuple with the use of function:
('G', 'e', 'e', 'k', 's')

Tuple with nested tuples:
((0, 1, 2, 3), ('python', 'geek'))
```

Accessing elements of Tuple

In order to access the tuple items refer to the index number. Use the index operator [ ] to access an item in a tuple. The index must be an integer. Nested tuples are accessed using nested indexing.

```
tuple1 = tuple([1, 2, 3, 4, 5])

# Accessing element using indexing
print("Frist element of tuple")
print(tuple1[0])

# Accessing element from last
# negative indexing
print("\nLast element of tuple")
print(tuple1[-1])

print("\nThird last element of tuple")
print(tuple1[-3])
```

Output:

```
Frist element of tuple
1

Last element of tuple
5

Third last element of tuple
3
```

Boolean

Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false. It is denoted by

the class bool.
Note – True and False with capital ‘T’ and ‘F’ are valid booleans otherwise python will throw an error.

```
print(type(True))
print(type(False))
```

Output:

```
<class 'bool'>
<class 'bool'>
```

Set

In Python, Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

Creating Sets

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by ‘comma’. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```
# Creating a Set

set1 = set()
print("Intial blank Set: ")
print(set1)

# Creating a Set with
# the use of a String
set1 = set("GeeksForGeeks")
print("\nSet with the use of String: ")
print(set1)

# Creating a Set with
# the use of a List
set1 = set(["Geeks", "For", "Geeks"])
```

```
print("\nSet with the use of List: ")
print(set1)

# Creating a Set with
# a mixed type of values
# (Having numbers and strings)
set1 = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])
print("\nSet with the use of Mixed Values")
print(set1)
```

Output:

```
Intial blank Set:
set()

Set with the use of String:
{'F', 'o', 'G', 's', 'r', 'k', 'e'}

Set with the use of List:
{'Geeks', 'For'}

Set with the use of Mixed Values
{1, 2, 4, 6, 'Geeks', 'For'}
```

Accessing elements of Sets

Set items cannot be accessed by referring to an index, since sets are unordered the items has no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
# Creating a set
set1 = set(["Geeks", "For", "Geeks"])
print("\nInitial set")
print(set1)

# Accessing element using
# for loop
print("\nElements of set: ")
for i in set1:
   print(i, end =" ")

# Checking the element
# using in keyword
```

```
print("Geeks" in set1)
```

Output:

```
Initial set:
{'Geeks', 'For'}

Elements of set:
Geeks For

True
```

Dictionary

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a 'comma'.

Creating Dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing it to curly braces{}.

Note – Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.

```
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
```

```
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with the use of Mixed Keys:
{1: [1, 2, 3, 4], 'Name': 'Geeks'}

Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}
```

Accessing elements of Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. There is also a method called get() that will also help in accessing the element from a dictionary.

```
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}
# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])
```

```
# accessing a element using get()
# method
print("Accessing a element using get:")
print(Dict.get(3))
```

Output:

```
Accessing a element using key:
For
Accessing a element using get:
Geeks
```

Input and Output statements:

Taking Input from the user (Input):

Sometimes a developer might want to take input from the user at some point in the program. To do this Python provides an input() function.

Syntax:

```
input('prompt')
```

where, prompt is a string that is displayed on the string at the time of taking input.

Example 1: Taking input from the user with a message.

```
# Taking input from the user
name = input("Enter your name: ")

 # Output
print("Hello, " + name)
```

Output:

```
Enter your name: Gfg

Hello, Gfg
```

Example 2: By default **input()** function takes the user's input in a string. So, to take the input in the form of int, you need to use **int()** along with input function.

```
# Taking input from the user as integer
num = int(input("Enter a number: "))
add = num + 1
```

```
# Output
print(add)
```

Output:

```
Enter a number: 25
26
```

Displaying Output (Output):

Python provides the print() function to display output to the console.

Syntax:

print(value(s), sep= ‘ ‘, end = ‘\n’, file=file, flush=flush)

Parameters:

value(s) : *Any value, and as many as you like. Will be converted to string before printed*

sep=’separator’ : *(Optional) Specify how to separate the objects, if there is more than one.Default :’ ‘*

end=’end’: *(Optional) Specify what to print at the end.Default : ‘\n’*

file : *(Optional) An object with a write method. Default :sys.stdout*

flush : *(Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False*

Returns: *It returns output to the screen.*

```
# Python program to demonstrate
# print() method
print("GFG")

# code for disabling the softspace feature
print('G', 'F', 'G', sep ='')

# using end argument
print("Python", end = '@')
print("GeeksforGeeks")
```

Output:

```
GFG
```

GFG

Python@GeeksforGeeks

Formatting Output

Formatting output in Python can be done in many ways. Let's discuss them below

- We can use **formatted string literals**, by starting a string with f or F before opening quotation marks or triple quotation marks. In this string, we can write Python expressions between **{** and **}** that can refer to a variable or any literal value.

Example:

```
# Declaring a variable
name = "Gfg"

# Output
print(f'Hello {name}! How are you?')
```

Output:

Hello Gfg! How are you?

- We can also use **format()** function to format our output to make it look presentable. The curly braces **{ }** work as placeholders. We can specify the order in which variables occur in the output.

Example:

```
a = 20
b = 10

# addition
sum = a + b

# subtraction
sub = a- b

# Output
print('The value of a is {} and b is {}'.format(a,b))

print('{2} is the sum of {0} and {1}'.format(a,b,sum))
print('{sub_value} is the subtraction of {value_a} and {value_b}'.format(value_a = a ,value_b =
b,sub_value = sub))
```

Output:

The value of a is 20 and b is 10

30 is the sum of 20 and 10

10 is the subtraction of 20 and 10

- We can use **'%'** operator. % values are replaced with zero or more value of elements.

Example:

```
num = int(input("Enter a value: "))

add = num + 5

# Output
print("The sum is %d" %add)
```

Output:

Enter a value: 50

The sum is 55

Control Statements:

As you may know, loops in Python are used to iterate repeatedly over a block of code. But at times, you might want to **shift the control once a particular condition is satisfied**. This is where control statements in Python come into the picture.

Control statements in python are **used to control the flow of execution of the program based on the specified conditions**. Python supports **3 control statements** such as,

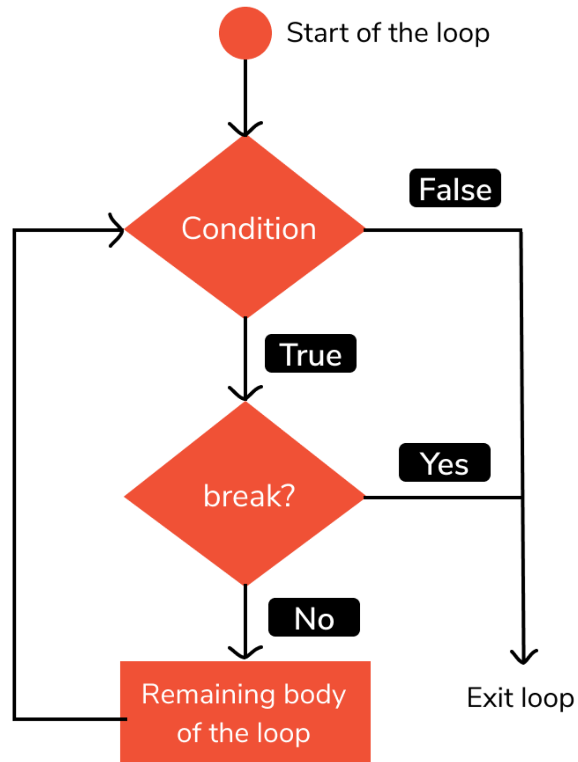
- Break
- Continue
- Pass

Break:

The break statement in Python is used to terminate a loop. This means whenever the interpreter encounters the break keyword, **it simply exits out of the loop**. Once it breaks out of the loop, the control shifts to the immediate next statement.

Also, if the break statement is used inside a nested loop, it terminates the innermost loop and the control shifts to the next statement in the outer loop.

Flowchart of Break Statement in Python



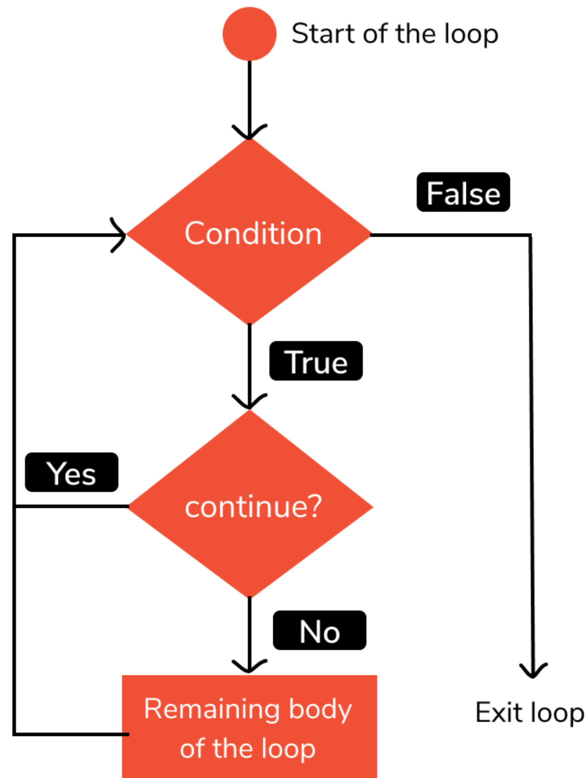
Example

```
#program to check if letter 'A' is present in the input

a = input ("Enter a word")
for i in a:
  if (i == 'A'):
    print ("A is found")
    break
 else:
    print ("A not found")

Input: FACE Prep

Output:
                A not found
                A is found
```

Continue:

Whenever the interpreter encounters a continue statement in Python, **it will skip the execution of the rest of the statements in that loop and proceed with the next iteration**. This means it returns the control to the beginning of the loop. Unlike the break statement, continue statement does not terminate or exit out of the loop. Rather, it continues with the next iteration. Here is the flow of execution when continue statement is used.

Flowchart of Continue Statement in Python



Example

```
#program to check if letter 'A' is present in the input
a = input ("Enter a word")
for i in a:
 if (i != 'A'):
  continue
 else:
  print ("A is found")


Input: FACE Prep
Output:
```

```
            A is found
```

Pass:

Assume we have a loop that is not implemented yet, but needs to be implemented in the future. In this case, if you leave the loop empty, the interpreter will throw an error. To avoid this, you can use the pass statement to construct a block that does nothing i.e contains no statements.

Example

```
for i in 'FACE':
  if (i == 'A'):
     pass
 print (i)

Output:
            F
            A
            C
            E
```

Operators:

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Python divides the operators in the following groups:

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators
- Assignment operators
- Identity operators
- Membership operators

Arithmetic operators:

Arithmetic operators are used with numeric values to perform common mathematical operations.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y+ 2
-	Subtract right operand from the left or unary minus	x - y- 2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Comparison operators:

Comparison operators are used to compare values. It returns either True or False according to the condition.

Operator	Meaning	Example
>	Greater than - True if left operand is greater than	x > y

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

Logical operators:

Operator	Meaning	Example
And	True if both the operands are true	x and y
Or	True if either of the operands is true	x or y
Not	True if operand is false (complements the operand)	not x

Bitwise operators:

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

Assignment operators

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x – 5
*=	x *= 5	x = x * 5

Identity operators:

is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

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

Membership operators:

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

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

Basic String operations:

String class i.e str provides many useful methods to manipulate string. Specifically, we will discuss methods which does the following.

1. Search for substring inside string.
2. Test strings.
3. Format strings.
4. Convert strings.

Recall from the earlier chapter that methods are functions which belongs to an object. However, unlike function, a method are always called on a object using the following notation.

object.method_name(arg1, arg2, arg3, ..., argN)

1. Searching and Replacing Strings:

The str class has the following methods which allow you to search for a substring inside a string.

Method	**Description**
endswith(sub)	Returns True if string ends with substring sub. Otherwise False.
startswith(sub)	Returns True if string starts with substring sub. Otherwise False.

Method	Description
find(sub)	Returns the lowest index of the string where substring sub is found. If substring sub is not found -1 is returned.
rfind(sub)	Returns the highest index of the string where substring sub is found. If substring sub is not found -1 is returned.
count(sub)	It returns the number of occurrences of substring sub found in the string. If no occurrences found 0 is returned.
replace(old, new)	It returns a new string after replacing old substring with new. Notice that it does not change the object on which it is called.

3. Formatting Methods:

The following table list some formatting methods of the str class.

Method	Description
center(width)	Returns a new copy of the string after centering it in a field of length width.
ljust(width)	Returns a new copy of the string justified to left in field of length width.
rjust(width)	Returns a new copy of the string justified to right in field of length width.

4. Converting Strings:

The following methods are commonly used to return a modified version of the string.

Method	Description
lower()	Returns a new copy of the string after converting all of it's characters to lowercase.
upper()	Returns a new copy of the string after converting all of it's characters to uppercase.
capitalize()	Returns a new copy of the string after capitalizing only the first letter in the string.
title()	Returns a new copy of the string after capitalizing the first letter in each word.
swapcase()	Returns a new copy after converting lowercase letters to uppercase and vice-versa.
strip()	Returns a new copy of the string after removing all the leading and trailing whitespace characters.
strip(chars)	Returns a new copy of the string after removing chars from the beginning and end of the string.

String Testing methods:

The following methods of the str class tests various types of characters inside the string.

Method	Description
str.isalnum()	returns True if all the characters in the string is alphanumeric (a string which contains either number or alphabets or both). Otherwise False.
str.isalpha()	returns True if all the characters in the string are alphabets. Otherwise False.
str.isdigit()	returns True if all the characters in the string are digits. Otherwise False.
str.islower()	returns True if all the characters in the string are in lowercase. Otherwise False.
str.isupper()	returns True if all the characters in the string are in uppercase. Otherwise False.
str.isspace()	returns True if all the characters in the string are whitespace characters. Otherwise False.

List:

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list

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

Tuple:

Python has two built-in methods that you can use on tuples.

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

Dictionary:

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value

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