Python

Introduction: Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum in early 1990's.It's a general public licenced software.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. and it has fewer syntactical constructions than other languages.

Features of python :

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Along with the above-mentioned features, Python has a big list of good features, few are listed below −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Getting Python :

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python https://www.python.org/

You can download Python documentation from https://www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats.

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example : a=10 integer 10 will be assigned to a variable a.

A=10 integer

B='Thub' string

C=10.879 float

Python allows you to assign a single value to several variables simultaneously.

For example −

A=b=c=1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.
For example –

a,b,c=1,23.5,'Thub'

Here are some examples of numbers −

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Operators :

Operators are the constructs which can manipulate the value of operands.

Python language supports the following types of operators.

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

1. Arithmetic Operators

Arithmetic Operators can be used to perform mathametical operations like addition, subtraction

Operator	Task	Example
+	Add two operands	x + y
-	Subtract right operand from the left	x - y
*	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
//	Floor division - division that results into whole number	x // y
**	Exponent - left operand raised to the power of right	x**y

2. logical operators

used to perform logical operations like and,or,not

Operator	Task	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	not x

3. comparision operators

used to compare values on either side and returns True or False

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

4. Assignment Operators

used to assign values to variables

Operator	Example	Task
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5

*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

5.Bitwise operators

used to perform operations bit by bit

a=10,b=12

binary of a=1010,b=1100

Operator	Task	Example
&	bitwise and	a&b=8
|	bitwise or	a|b=15
^	bitwise XOR	a^b=6
~	Complement	~a= (-11)
<<	Bitwise left shift	a<<2 =40
>>	Bitwise right shift	a>>2 =2

6.membership operator

'in' and 'not in' are the membership operators in Python.used to

check particular values in core data types like lists. and returns

True/False

let x="python"

Operator	Task	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

7.Identity operator

'is' and 'is not' are the identity operators in Python.

hey are used to check if two variables are having

the same part of the memory.Two variables that are

equal does not imply that they are identical.and returns True or False

let a='python',b='codemind'

Operator	Task	Example
is	True if the operands are identical	a is b (False)
is not	True if the operands are not identical	a is not b (True)

Conditional Statements:

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

Indentation in Python

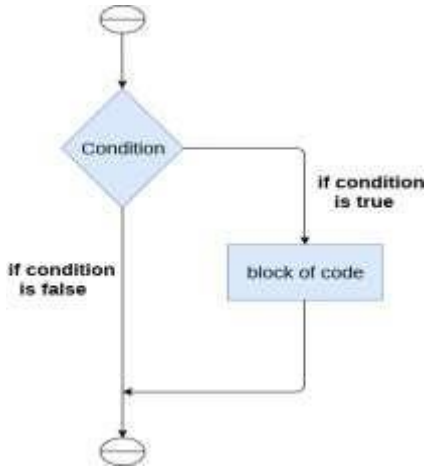
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or flase



The syntax of the if-statement is given below.

```
if expression:
    statement
```

Example:

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
```

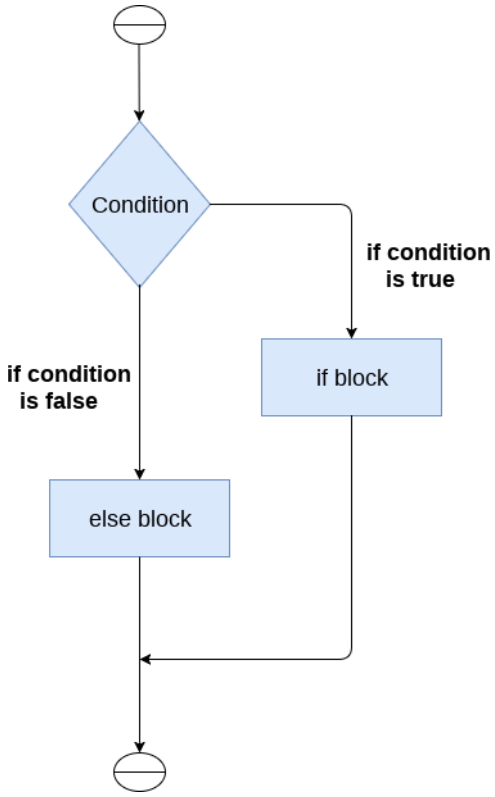
output:

```
enter the number?10
Number is even
```

The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below.

```
if condition:
   #block of statements
else:
    #another block of statements (else-block)
```

Example:

```
age = int (input("Enter your age? "))
if age>=18:
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

Ouput:

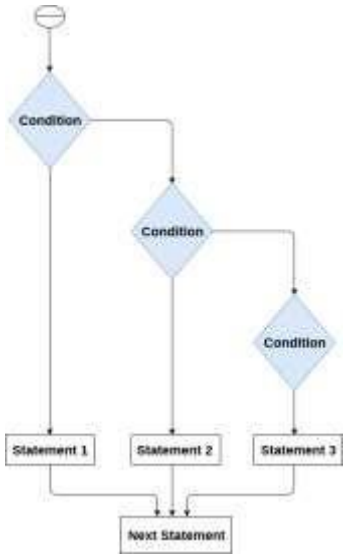
```
Enter your age? 90
You are eligible to vote !!
```

The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement is given below.




```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```

Eaxmple:

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

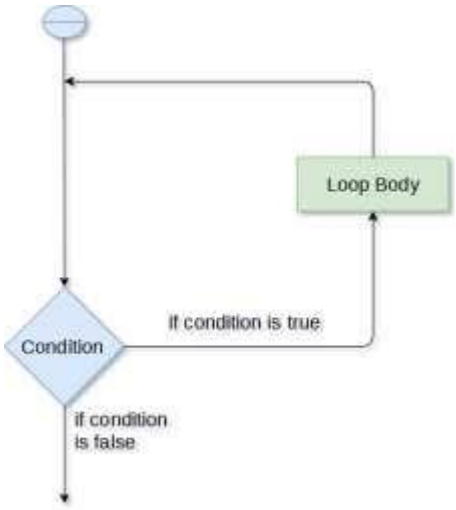
output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

Loops:

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantages of loops

There are the following advantages of loops in Python.

1. It provides code re-usability.
2. Using loops, we do not need to write the same code again and again.
3. Using loops, we can traverse over the elements of data structures (array or linked lists).

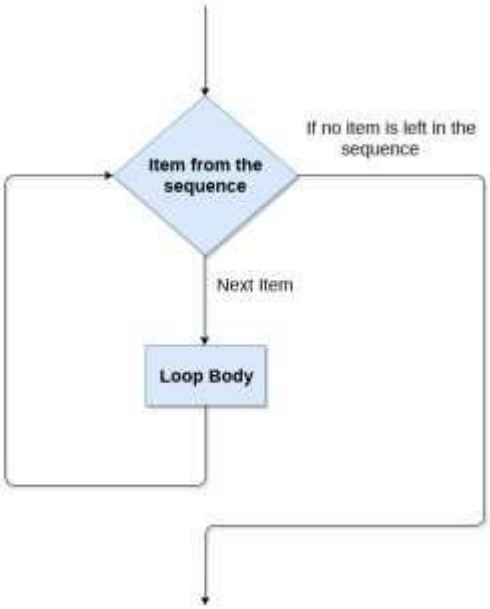
There are the following loop statements in Python.

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

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

The for loop flowchart



For loop Using range() function

The range() function

The **range()** function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9. The syntax of the range() function is given below.

Syntax:

range(start,stop,step size)

- The start represents the beginning of the iteration.
- The stop represents that the loop will iterate till stop-1. The **range(1,5)** will generate numbers 1 to 4 iterations. It is optional.
- The step size is used to skip the specific numbers from the iteration. It is optional to use. By default, the step size is 1. It is optional.

Consider the following examples:

Example: Program to print numbers in sequence.

```
for i in range(10):
    print(i,end = ' ')
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

Using else statement with for loop

Unlike other languages like C, C++, or Java, Python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

Example:

```
for i in range(0,5):
    print(i)
else:
    print("for loop completely exhausted, since there is no break.")
```

Output:

```
0
1
2
3
4
for loop completely exhausted, since there is no break.
```

The for loop completely exhausted, since there is no break.

While loop

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.

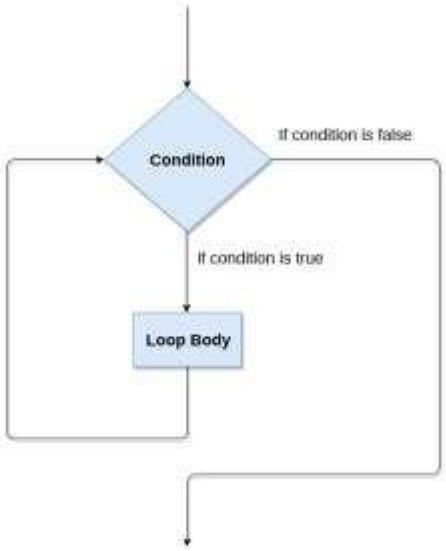
It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

The syntax is given below.

```
while expression:
    statements
```

Here, the statements can be a single statement or a group of statements. The expression should be any valid Python expression resulting in true or false. The true is any non-zero value and false is 0.

While loop Flowchart



Example-1: Program to print 1 to 10 using while loop

```
i=1
#The while loop will iterate until condition becomes false.
While(i<=10):
    print(i)
    i=i+1
```

Output:

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

Infinite while loop

If the condition is given in the while loop never becomes false, then the while loop will never terminate, and it turns into the **infinite while loop.**

Any **non-zero** value in the while loop indicates an **always-true** condition, whereas zero indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Example

```
while (1):
    print("Hi! we are inside the infinite while loop")
```

Output:

```
Hi! we are inside the infinite while loop
Hi! we are inside the infinite while loop
……
```

Using else with while loop

Python allows us to use the else statement with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed, and the statement present after else block will be executed. The else statement is optional to use with the while loop. Consider the following example.

Example 1

```
i=1
while(i<=5):
    print(i)
    i=i+1
else:
    print("The while loop exhausted")
```

break statement

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

The syntax of the break is given below.

```
#loop statements
break;
```

Example : break statement with while loop

```
i = 0;
while 1:
    print(i," ",end=""),
    i=i+1;
    if i == 10:
        break;
print("came out of while loop");
```

Output:

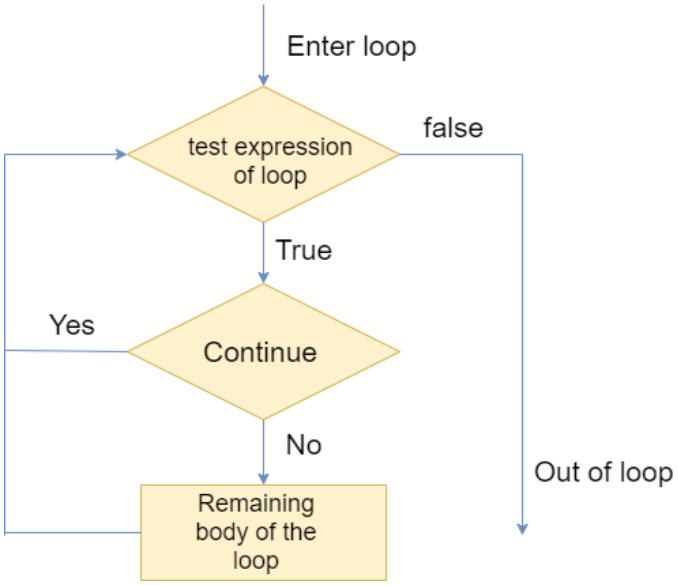
```
0  1  2  3  4  5  6  7  8  9  came out of while loop
```

continue Statement

The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

Syntax

```
#loop statements
continue
#the code to be skipped
```



Consider the following examples.

Example 1

```
i = 0
while(i < 10):
   i = i+1
   if(i == 5):
      continue
   print(i)
```

Output:

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

Observe the output of above code, the value 5 is skipped because we have provided the **if condition** using with **continue statement** in while loop. When it matched with the given condition then control transferred to the beginning of the while loop and it skipped the value 5 from the code.

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example

```
def my_function():
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Example

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

now this function gets an error.

Types of Arguments :

1,Position based Arguments

2,Key word based Arguments

3,Defult based Arguments

4,Veriable length Argument

Position Based Arguments:

Example

```
def person(name , age):
        print(name)
        print(age)
person('Ashok',26)
```

we are passing Argument based on position since the first position is name so we are passing ‘Ashok’ and the second position is age so that we are passing 26

Key word Based Arguments:

Example

```
def person(name , age):
    print(name)
    print(age)
```

```
person(age=25,name='Ashok')
```

As we don’t Know the exact positions then we use keyword to pass the Arguments

If we observe clearly in the call of function, we are passing arguments based on their respective keywords.

Default Based Arguments:

Example

```
def person(name , age=18):
    print(name)
    print(age)
person('Ashok',25)
```

output:

Ashok

25

if we notice in the function declaration, one of the argument is declared with a default value, so even though if you don’t pass the age value while calling the function it doesn’t give any error because the age is given defaultly

Example

```
def person(name , age=18):
    print(name)
    print(age)
person('Ashok')
```

output:

Ashok

18

It get executed with the default value in the above example

Variable Length Argument:

This function expects 2 arguments, and gets 2 arguments:

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Example:

```
def sum(*b):
        c = 0
        for i in b:
        c = c + i
        print(c)
sum(5,6,34,78)
```

while we are using variable-length arguments all the multiple value passed were passed while calling function is read as tuple.

In the above example all the values stored in tuble, i,e b is a tuble

b=(5,6,34,78)

Keyworded Variable Length Arguments:

The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

A keyword argument is where you provide a name to the variable as you pass it into the function.

One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn't seem to be any order in which they were printed out.

Example

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print (key, value)
```

```
# Driver code

myFun(first ='Geeks', mid ='for', last='Geeks')
```

while using Keyworded Variable Length Arguments the data is read in form of dictionary here keyword is a key and data is value.

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return Values

To let a function return a value, use the return statement:

Example

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def myfunction():
  pass
```

Python Scope of Variable in Functions

A variable is only available from inside the region it is created. This is called **scope**.

Local Scope

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Example

A variable created inside a function is available inside that function:

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

Function Inside Function

As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function:

Example

The local variable can be accessed from a function within the function

```
def myfunc():
  x = 300
  def myinnerfunc():
  print(x)
  myinnerfunc()

myfunc()
```

Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example

A variable created outside of a function is global and can be used by anyone:

```
x = 300

def myfunc ():
    print(x)

myfunc()

print(x)
```

Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example

The function will print the local x, and then the code will print the global x:

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the global keyword.

The global keyword makes the variable global.

Example

If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

Also, use the global keyword if you want to make a change to a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

core data types:

list,set,tuple and dictionary are the 4 core data types in python,which works similarly like arrays but the advantage with this core data types is to store hetrogeniuos data type at a time. All of these four have their individual functionolity.

lists

Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Consider the following example.

```
list1  = [1, "hi", "Python", 2]

#Checking type of given list

print(type(list1))

#Printing the list1

print (list1)
```

Characteristics of Lists

The list has the following characteristics:

The lists are ordered.

The element of the list can access by index.

The lists are the mutable type.

The lists are mutable types.

A list can store the number of various elements.

Let's check the first statement that lists are the ordered.

```
a = [1,2,"Peter",4.50,"Ricky",5,6]

b = [1,2,5,"Peter",4.50,"Ricky",6]

a ==b
```

Output:

```
False
```

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of

objects.

```
a = [1, 2,"Peter", 4.50,"Ricky",5, 6]

b = [1, 2,"Peter", 4.50,"Ricky",5, 6]

a == b
```

Output:

```
True
```

Let's have a look at the list example in detail.

```
emp = ["John", 102, "USA"]

Dep1 = ["CS",10]

Dep2 = ["IT",11]

HOD_CS = [10,"Mr. Holding"]

HOD_IT = [11, "Mr. Bewon"]

print("printing employee data...")

print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))

print("printing departments...")

print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID:
%s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]))

print("HOD Details ....")

print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]))

print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]))

print(type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT))
```

Output:

```
printing employee data...

Name : John, ID: 102, Country: USA
```

printing departments...

Department 1:

Name: CS, ID: 11

Department 2:

Name: IT, ID: 11

HOD Details ....

CS HOD Name: Mr. Holding, Id: 10

IT HOD Name: Mr. Bewon, Id: 11

<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>

In the above example, we have created the lists which consist of the employee and department details and printed the corresponding details. Observe the above code to understand the concept of the list better.

List indexing and splitting

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

We can get the sub-list of the list using the following syntax.

list_varible(start:stop:step)

The start denotes the starting index position of the list.

The stop denotes the last index position of the list.

The step is used to skip the nth element within a start:stop

Consider the following example:

list = [1,2,3,4,5,6,7]

```
print(list[0])

print(list[1])

print(list[2])

print(list[3])

# Slicing the elements

print(list[0:6])

# By default the index value is 0 so its starts from the 0th element and go for index -
1.

print(list[:])

print(list[2:5])

print(list[1:6:2])
```

Output:

```
1

2

3

4

[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 7]

[3, 4, 5]

[2, 4, 6]
```

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on

until the left-most elements are encountered.

Let's have a look at the following example where we will use negative indexing to access the elements of the list.

```
list = [1,2,3,4,5]

print(list[-1])

print(list[-3:])

print(list[:-1])

print(list[-3:-1])
```

Output:

```
5

[3, 4, 5]

[1, 2, 3, 4]

[3, 4]
```

As we discussed above, we can get an element by using negative indexing. In the above code, the first print statement returned the rightmost element of the list. The second print statement returned the sub-list, and so on.

Updating List values

Lists are the most versatile data structures in Python since they are mutable, and their values can be updated by using the slice and assignment operator.

Python also provides append() and insert() methods, which can be used to add values to the list.

Consider the following example to update the values inside the list.

```
list = [1, 2, 3, 4, 5, 6]

print(list)

# It will assign value to the value to the second index
```

```
list[2] = 10

print(list)

# Adding multiple-element

list[1:3] = [89, 78]

print(list)

# It will add value at the end of the list

list[-1] = 25

print(list)
```

Output:

```
[1, 2, 3, 4, 5, 6]

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

[1, 89, 78, 4, 5, 25]
```

The list elements can also be deleted by using the del keyword. Python also provides us the remove() method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

```
list = [1, 2, 3, 4, 5, 6]

print(list)

# It will assign value to the value to second index

list[2] = 10

print(list)

# Adding multiple element
```

```
list[1:3] = [89, 78]

print(list)

# It will add value at the end of the list

list[-1] = 25

print(list)
```

Output:

```
[1, 2, 3, 4, 5, 6]

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

[1, 89, 78, 4, 5, 25]
```

Python List Operations

The concatenation (+) and repetition (*) operators work in the same way as they were working with the strings.

Let's see how the list responds to various operators.

Consider a Lists l1 = [1, 2, 3, 4], and l2 = [5, 6, 7, 8] to perform operation.

Repetition The repetition operator enables the list elements to be repeated multiple times.

L1*2 = [1, 2, 3, 4, 1, 2, 3, 4]

Concatenation It concatenates the list mentioned on either side of the operator.

l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8]

Membership It returns true if a particular item exists in a particular list otherwise false.

print(2 in l1) prints True.

Iteration The for loop is used to iterate over the list elements.

```
for i in l1:
    print(i)
```

Output

1

2

3

4

Length It is used to get the length of the list

len(l1) = 4

Iterating a List

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

```
list = ["John", "David", "James", "Jonathan"]
for i in list:
    # The i variable will iterate over the elements of the List and contains each element in each iteration.
    print(i)
```

Output:

John

David

James

Jonathan

Adding elements to the list

Python provides append() function which is used to add an element to the list. However, the append() function can only add value to the end of the list.

Consider the following example in which, we are taking the elements of the list from the user and printing the list on the console.

```
#Declaring the empty list

l =[]

#Number of elements will be entered by the user

n = int(input("Enter the number of elements in the list:"))

# for loop to take the input

for i in range(0,n):

    # The input is taken from the user and added to the list as the item

    l.append(input("Enter the item:"))

print("printing the list items..")

# traversal loop to print the list items

for i in l:

    print(i, end = "  ")
```

Output:

```
Enter the number of elements in the list:5

Enter the item:25

Enter the item:46

Enter the item:12

Enter the item:75

Enter the item:42
```

printing the list items

25 46 12 75 42

Removing elements from the list

Python provides the remove() function which is used to remove the element from the list. Consider the following example to understand this concept.

Example -

```
list = [0,1,2,3,4]

print("printing original list: ");

for i in list:

    print(i,end=" ")

list.remove(2)

print("\nprinting the list after the removal of first element...")

for i in list:

    print(i,end=" ")
```

Output:

printing original list:

0 1 2 3 4

printing the list after the removal of first element...

0 1 3 4

Python List Built-in functions

Python provides the following built-in functions, which can be used with the lists.

1 cmp(list1, list2) It compares the elements of both the lists. This method is not used in the Python 3 and the above versions.

2 len(list) It is used to calculate the length of the list.

```
L1 = [1,2,3,4,5,6,7,8]
print(len(L1))
```

8

3 max(list) It returns the maximum element of the list.

```
L1 = [12,34,26,48,72]
print(max(L1))
```

72

4 min(list) It returns the minimum element of the list.

```
L1 = [12,34,26,48,72]
print(min(L1))
```

12

5 list(seq) It converts any sequence to the list.

```
str = "Johnson"
s = list(str)
print(type(s))
```

<class list>

Let's have a look at the few list examples.

Example: 1- Write the program to remove the duplicate element of the list.

```
list1 = [1,2,2,3,55,98,65,65,13,29]
# Declare an empty list that will store unique values
list2 = []
```

```
for i in list1:
    if i not in list2:
        list2.append(i)
print(list2)
```

Output:

[1, 2, 3, 55, 98, 65, 13, 29]

Example:2- Write a program to find the sum of the element in the list.

```
list1 = [3,4,5,9,10,12,24]
sum = 0
for i in list1:
    sum = sum+i
print("The sum is:",sum)
```

Output:

The sum is: 67

Example: 3- Write the program to find the lists consist of at least one common element.

```
list1 = [1,2,3,4,5,6]
list2 = [7,8,9,2,10]
for x in list1:
    for y in list2:
        if x == y:
            print("The common element is:",x)
```

Output:

The common element is: 2

Tuple

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

Creating a tuple

A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The parentheses are optional but it is good practice to use. A tuple can be defined as follows.

T1 = (101, "Peter", 22)

T2 = ("Apple", "Banana", "Orange")

T3 = 10,20,30,40,50

print(type(T1))

print(type(T2))

print(type(T3))

Output:

<class 'tuple'>

<class 'tuple'>

<class 'tuple'>

Note: The tuple which is created without using parentheses is also known as tuple packing.

An empty tuple can be created as follows.

T4 = ()

Creating a tuple with single element is slightly different. We will need to put comma

after the element to declare the tuple.

```
tup1 = ("technicalhub")

print(type(tup1))

#Creating a tuple with single element

tup2 = ("technicalhub",)

print(type(tup2))
```

Output:

```
<class 'str'>

<class 'tuple'>
```

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

Consider the following example of tuple:

Example - 1

```
tuple1 = (10, 20, 30, 40, 50, 60)

print(tuple1)

count = 0

for i in tuple1:

    print("tuple1[%d] = %d"%(count, i))

    count = count+1
```

Output:

```
(10, 20, 30, 40, 50, 60)

tuple1[0] = 10

tuple1[1] = 20
```

tuple1[2] = 30

tuple1[3] = 40

tuple1[4] = 50

tuple1[5] = 60

Example - 2

```
tuple1 = tuple(input("Enter the tuple elements ..."))

print(tuple1)

count = 0

for i in tuple1:

    print("tuple1[%d] = %s"%(count, i))

    count = count+1
```

Output:

Enter the tuple elements ...123456

('1', '2', '3', '4', '5', '6')

tuple1[0] = 1

tuple1[1] = 2

tuple1[2] = 3

tuple1[3] = 4

tuple1[4] = 5

tuple1[5] = 6

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

We will see all these aspects of tuple in this section of the tutorial.

Tuple indexing and slicing

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.

Consider the following example:

```
tup = (1,2,3,4,5,6,7)

print(tup[0])

print(tup[1])

print(tup[2])

# It will give the IndexError

print(tup[8])
```

Output:

```
1

2

3

tuple index out of range
```

In the above code, the tuple has 7 elements which denote 0 to 6. We tried to access an element outside of tuple that raised an IndexError.

```
tuple = (1,2,3,4,5,6,7)

#element 1 to end

print(tuple[1:])

#element 0 to 3 element
```

```
print(tuple[:4])

#element 1 to 4 element

print(tuple[1:5])

# element 0 to 6 and take step of 2

print(tuple[0:6:2])
```

Output:

```
(2, 3, 4, 5, 6, 7)

(1, 2, 3, 4)

(1, 2, 3, 4)

(1, 3, 5)
```

Negative Indexing

The tuple element can also access by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second last item and so on.

The elements from left to right are traversed using the negative indexing. Consider the following example:

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

print(tuple1[-1])

print(tuple1[-4])

print(tuple1[-3:-1])

print(tuple1[:-1])

print(tuple1[-2:])
```

Output:

```
5
```

2

(3, 4)

(1, 2, 3, 4)

(4, 5)

Deleting Tuple

Unlike lists, the tuple items cannot be deleted by using the del keyword as tuples are immutable. To delete an entire tuple, we can use the del keyword with the tuple name.

Consider the following example.

```
tuple1 = (1, 2, 3, 4, 5, 6)

print(tuple1)

del tuple1[0]

print(tuple1)

del tuple1

print(tuple1)
```

Output:

```
(1, 2, 3, 4, 5, 6)

Traceback (most recent call last):

  File "tuple.py", line 4, in <module>

    print(tuple1)

NameError: name 'tuple1' is not defined
```

Basic Tuple operations

The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

Repetition The repetition operator enables the tuple elements to be repeated multiple times.

T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

Concatenation It concatenates the tuple mentioned on either side of the operator.

T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9)

Membership It returns true if a particular item exists in the tuple otherwise false

print (2 in T1) prints True.

Iteration The for loop is used to iterate over the tuple elements.

```
for i in T1:
    print(i)
```

Output

1

2

3

4

5

Length It is used to get the length of the tuple.

len(T1) = 5

Python Tuple inbuilt functions

1 cmp(tuple1, tuple2) It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.

2 len(tuple) It calculates the length of the tuple.

3	max(tuple)	It returns the maximum element of the tuple
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.

Where use tuple?

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.

2. Tuple can simulate a dictionary without keys. Consider the following nested structure, which can be used as a dictionary.

[(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]

Set:

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.

Creating a set

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

Example 1: Using curly braces

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}

print(Days)

print(type(Days))
```

```
print("looping through the set elements ... ")

for i in Days:

    print(i)
```

Output:

```
{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}

<class 'set'>

looping through the set elements ...

Friday

Tuesday

Monday

Saturday

Thursday

Sunday

Wednesday
```

Example 2: Using set() method

```
Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
"Sunday"])

print(Days)

print(type(Days))

print("looping through the set elements ... ")

for i in Days:

    print(i)
```

Output:

{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}

<class 'set'>

looping through the set elements ...

Friday

Wednesday

Thursday

Saturday

Monday

Tuesday

Sunday

It can contain any type of element such as integer, float, tuple etc. But mutable elements (list, dictionary, set) can't be a member of set. Consider the following example.

```
# Creating a set which have immutable elements

set1 = {1,2,3, "JavaTpoint", 20.5, 14}

print(type(set1))

#Creating a set which have mutable element

set2 = {1,2,3,["Javatpoint",4]}

print(type(set2))
```

Output:

<class 'set'>

Traceback (most recent call last)

<ipython-input-5-9605bb6fbc68> in <module>

```
    4

    5 #Creating a set which holds mutable elements

----> 6 set2 = {1,2,3,["Javatpoint",4]}

    7 print(type(set2))

TypeError: unhashable type: 'list'
```

In the above code, we have created two sets, the set set1 have immutable elements and set2 have one mutable element as a list. While checking the type of set2, it raised an error, which means set can contain only immutable elements.

Creating an empty set is a bit different because empty curly {} braces are also used to create a dictionary as well. So Python provides the set() method used without an argument to create an empty set.

```
# Empty curly braces will create dictionary

set3 = {}

print(type(set3))

# Empty set using set() function

set4 = set()

print(type(set4))
```

Output:

```
<class 'dict'>

<class 'set'>
```

Let's see what happened if we provide the duplicate element to the set.

```
set5 = {1,2,4,4,5,8,9,9,10}

print("Return set with unique elements:",set5)
```

Output:

Return set with unique elements: {1, 2, 4, 5, 8, 9, 10}

In the above code, we can see that set5 consisted of multiple duplicate elements when we printed it remove the duplicity from the set.

Adding items to the set

Python provides the add() method and update() method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set. Consider the following example.

Example: 1 - Using add() method

```
Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(months)

print("\nAdding other months to the set...");

Months.add("July");

Months.add ("August");

print("\nPrinting the modified set...");

print(Months)

print("\nlooping through the set elements ... ")

for i in Months:

    print(i)
```

Output:

```
printing the original set ...

{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...
```

Printing the modified set...

{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...

February

July

May

April

March

August

June

January

To add more than one item in the set, Python provides the update() method. It accepts iterable as an argument.

Consider the following example.

Example - 2 Using update() function

```
Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nupdating the original set ... ")

Months.update(["July","August","September","October"]);

print("\nprinting the modified set ... ")

print(Months);
```

Output:

printing the original set ...

{'January', 'February', 'April', 'May', 'June', 'March'}

updating the original set ...

printing the modified set ...

{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}

Removing items from the set

Python provides the discard() method and remove() method which can be used to remove the items from the set. The difference between these function, using discard() function if the item does not exist in the set then the set remain unchanged whereas remove() method will through an error.

Consider the following example.

Example-1 Using discard() method

```
months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(months)

print("\nRemoving some months from the set...");

months.discard("January");

months.discard("May");

print("\nPrinting the modified set...");

print(months)

print("\nlooping through the set elements ... ")

for i in months:

    print(i)
```

Output:

printing the original set ...

{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...

Printing the modified set...

{'February', 'March', 'April', 'June'}

looping through the set elements ...

February

March

April

June

Python provides also the remove() method to remove the item from the set. Consider the following example to remove the items using remove() method.

Example-2 Using remove() function

```
months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(months)

print("\nRemoving some months from the set...");

months.remove("January");

months.remove("May");

print("\nPrinting the modified set...");

print(months)
```

Output:

printing the original set ...

{'February', 'June', 'April', 'May', 'January', 'March'}

Removing some months from the set...

Printing the modified set...

{'February', 'June', 'April', 'March'}

We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped from set.

Consider the following example to remove the item from the set using pop() method.

```
Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nRemoving some months from the set...");

Months.pop();

Months.pop();

print("\nPrinting the modified set...");

print(Months)
```

Output:

printing the original set ...

{'June', 'January', 'May', 'April', 'February', 'March'}

Removing some months from the set...

Printing the modified set...

{'May', 'April', 'February', 'March'}

In the above code, the last element of the Month set is March but the pop() method removed the June and January because the set is unordered and the pop() method could not determine the last element of the set.

Python provides the clear() method to remove all the items from the set.

Consider the following example.

```
Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nRemoving all the items from the set...");

Months.clear()

print("\nPrinting the modified set...")

print(Months)
```

Output:

```
printing the original set ...

{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...

set()
```

Difference between discard() and remove()

Despite the fact that discard() and remove() method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the Python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the Python will raise an error.

Consider the following example.

Example-

```
Months = set(["January","February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")

print(Months)

print("\nRemoving items through discard() method...");

Months.discard("Feb"); #will not give an error although the key feb is not available in the set

print("\nprinting the modified set...")

print(Months)

print("\nRemoving items through remove() method...");

Months.remove("Jan") #will give an error as the key jan is not available in the set.

print("\nPrinting the modified set...")

print(Months)
```

Output:

```
printing the original set ...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through remove() method...

Traceback (most recent call last):

  File "set.py", line 9, in
```

```
Months.remove("Jan")
```

```
KeyError: 'Jan'
```

Dictionory:

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

Keys must be a single element

Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

Syntax:

```
Dict = {"Name": "Tom", "Age": 22}
```

In the above dictionary Dict, The keys Name and Age are the string that is an immutable object.

Let's see an example to create a dictionary and print its content.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print(Employee)
```

Output

<class 'dict'>

Printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Python provides the built-in function dict() method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

```
# Creating an empty Dictionary

Dict = {}

print("Empty Dictionary: ")

print(Dict)

# Creating a Dictionary

# with dict() method

Dict = dict({1: 'Java', 2: 'T', 3:'Point'})

print("\nCreate Dictionary by using  dict(): ")

print(Dict)

# Creating a Dictionary

# with each item as a Pair

Dict = dict([(1, 'Devansh'), (2, 'Sharma')])

print("\nDictionary with each item as a pair: ")

print(Dict)
```

Output:

Empty Dictionary:

{}

Create Dictionary by using dict():

{1: 'Java', 2: 'T', 3: 'Point'}

Dictionary with each item as a pair:

{1: 'Devansh', 2: 'Sharma'}

Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print("Name : %s" %Employee["Name"])

print("Age : %d" %Employee["Age"])

print("Salary : %d" %Employee["salary"])

print("Company : %s" %Employee["Company"])
```

Output:

```
<class 'dict'>

printing Employee data ....

Name : John

Age : 29

Salary : 25000
```

Company : GOOGLE

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

Adding dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key Dict[key] = value. The update() method is also used to update an existing value.

Note: If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

Let's see an example to update the dictionary values.

Example - 1:

```
# Creating an empty Dictionary

Dict = {}

print("Empty Dictionary: ")

print(Dict)

# Adding elements to dictionary one at a time

Dict[0] = 'Peter'

Dict[2] = 'Joseph'

Dict[3] = 'Ricky'

print("\nDictionary after adding 3 elements: ")

print(Dict)

# Adding set of values

# with a single Key

# The Emp_ages doesn't exist to dictionary
```

```
Dict['Emp_ages'] = 20, 33, 24

print("\nDictionary after adding 3 elements: ")

print(Dict)

# Updating existing Key's Value

Dict[3] = 'JavaTpoint'

print("\nUpdated key value: ")

print(Dict)
```

Output:

```
Empty Dictionary:

{}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:

{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Example - 2:

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print(Employee)

print("Enter the details of the new employee....");
```

```
Employee["Name"] = input("Name: ");

Employee["Age"] = int(input("Age: "));

Employee["salary"] = int(input("Salary: "));

Employee["Company"] = input("Company:");

print("printing the new data");

print(Employee)
```

Output:

```
Empty Dictionary:

{}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:

{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Deleting elements using del keyword

The items of the dictionary can be deleted by using the del keyword as given below.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print(type(Employee))

print("printing Employee data .... ")

print(Employee)

print("Deleting some of the employee data")
```

```
del Employee["Name"]

del Employee["Company"]

print("printing the modified information ")

print(Employee)

print("Deleting the dictionary: Employee");

del Employee

print("Lets try to print it again ");

print(Employee)
```

Output:

```
<class 'dict'>

printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Deleting some of the employee data

printing the modified information

{'Age': 29, 'salary': 25000}

Deleting the dictionary: Employee

Lets try to print it again

NameError: name 'Employee' is not defined
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

Using pop() method

The pop() method accepts the key as an argument and remove the associated value. Consider the following example.

```
# Creating a Dictionary

Dict = {1: 'Java', 2: 'Peter', 3: 'Thomas'}

# Deleting a key

# using pop() method

pop_ele = Dict.pop(3)

print(Dict)
```

Output:

{1: 'Java', 2: 'Peter'}

Python also provides a built-in methods popitem() and clear() method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

Iterating Dictionary

A dictionary can be iterated using for loop as given below.

Example 1

```
# for loop to print all the keys of a dictionary

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

for x in Employee:

    print(x)
```

Output:

Name

Age

salary

Company

Example 2

```
#for loop to print all the values of the dictionary
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee:
    print(Employee[x])
```

Output:

```
John
29
25000
GOOGLE
```

Example - 3

```
#for loop to print the values of the dictionary by using values() method.
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee.values():
    print(x)
```

Output:

```
John
29
25000
GOOGLE
```

Example 4

```
#for loop to print the items of the dictionary by using items() method.
```

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

for x in Employee.items():

    print(x)
```

Output:

```
('Name', 'John')

('Age', 29)

('salary', 25000)

('Company', 'GOOGLE')
```

Properties of Dictionary keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

```
Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}

for x,y in Employee.items():

    print(x,y)
```

Output:

```
Name John

Age 29

Salary 25000

Company GOOGLE
```

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Consider the following example.

```
Employee = {"Name": "John", "Age": 29,
"salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}

for x,y in Employee.items():

    print(x,y)
```

Output:

```
Traceback (most recent call last):

  File "dictionary.py", line 1, in

    Employee = {"Name": "John", "Age": 29,
"salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}

TypeError: unhashable type: 'list'
```

Built-in Dictionary functions

The built-in python dictionary methods along with the description are given below.

1	cmp(dict1, dict2)	It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false.
2	len(dict)	It is used to calculate the length of the dictionary.
3	str(dict)	It converts the dictionary into the printable string representation.
4	type(variable)	It is used to print the type of the passed variable.

Built-in Dictionary methods

The built-in python dictionary methods along with the description are given below.

1	dic.clear()	It is used to delete all the items of the dictionary.
2	dict.copy()	It returns a shallow copy of the dictionary.
3	dict.fromkeys(iterable, value = None, /)	Create a new dictionary from the iterable with the values equal to value.
4	dict.get(key, default = "None")	It is used to get the value specified for the passed key.
5	dict.has_key(key)	It returns true if the dictionary contains the specified key.
6	dict.items()	It returns all the key-value pairs as a tuple.
7	dict.keys()	It returns all the keys of the dictionary.
8	dict.setdefault(key,default= "None")	It is used to set the key to the default value if the key is not specified in the dictionary
9	dict.update(dict2)	It updates the dictionary by adding the key-value pair of dict2 to this dictionary.
10	dict.values()	It returns all the values of the dictionary.