**1Introduction to the print() function in python**

Python print() function prints the message to the screen or any other standard output device.

*# print() function example*

print("GeeksforGeeks")

a = [1, 2, 'gfg']

print(a)

In this example, we have 2 variables integer and [string](https://www.geeksforgeeks.org/python-string/). We are printing all variables with print() function.

name = "John"

age = 30

print("Name:", name)

print("Age:", age)

**2.Formatting outputs using f-strings and format()**

f-strings (formatted string literals) are a way to embed expressions inside string literals in Python, using curly braces {}. They provide an easy and readable way to format strings dynamically.

name = "Alice"  
age = 30  
sentence = f"My name is {name} and I am {age} years old."  
print(sentence)  
Output:  
My name is Alice and I am 30 years old.

**3. Using input() function to read user input from the keyword**

**Python input() function** is used to take user input. By default, it returns the user input in form of a string.

**Syntax:**

input(prompt)

**prompt [optional]:** any string value to display as input message

Ex: input(“What is your name? “)

**Returns:** Return a string value as input by the user.

**4.Converting user input into different data types(e. int ,float , etc)**

# Taking input as string

color = input("What color is rose?: ")

print(color)

# Taking input as int

# Typecasting to int

n = int(input("How many roses?: "))

print(n)

# Taking input as float

# Typecasting to float

price = float(input("Price of each rose?: "))

print(price)

**Output:**

What color is rose?: red  
red  
How many roses?: 10  
10  
Price of each rose?: 15.50  
15.5

**5.Opening files in different modes(‘r’,’w’,’a’,’r+’,’w+’)**

### **Append Mode in Python with ‘a’**

In ‘a’ mode, the file is opened for writing, positioned at the end if it exists, creates a new empty file if not, appends data without altering existing content, and disallows reading

with open('example.txt', 'a') as file:

    file.write('Appended text\n')

The ‘w’ is for write-only mode, and ‘w+’ is for both write and read mode. Use ‘w+’ if you need to both write and read data from the file, while ‘w’ is for writing data and overwriting the file’s contents.

### Write a file in Python with ‘w’

It opens the file for writing. If the file exists, it truncates (clears) its content. If the file doesn’t exist, Python creates a new, empty file. Reading from the file is not allowed in this mode.

with open('example.txt', 'w') as file:

    file.write('This will overwrite existing content')

### **Write and Read Mode in Python with ‘w+’**

In ‘w+’ mode, the file is opened for both reading and writing, existing content is cleared, a new empty file is created if it doesn’t exist, and the file pointer is positioned at the beginning.

* Python3

|  |
| --- |
| with open('example.txt', 'w+') as file:      file.write('This will overwrite existing content')      file.seek(0)  # Move the pointer to the beginning      content = file.read()      print(content) |

The ‘r’ is for reading only, and ‘r+’ is for both reading and writing. Be cautious when using ‘r+’ as it can potentially overwrite or modify the existing content of the file.

### **Read mode in Python with ‘r’**

This code segment opens a file named ‘file\_r.txt’ in ‘read’ mode (‘r’). It then reads the content of the file using the .read() method and stores it in the variable content. The ‘r’ mode is used for reading files, and it will raise a FileNotFoundError if the file does not exist.

with open('example.txt', 'r') as file\_r:

    content = file\_r.read()

    print('Content in "r":', content)

### **Read and Write Mode in Python with ‘r+’**

In ‘r+’ mode, the file is opened for both reading and writing without truncation, and the file pointer is positioned at the beginning. If the file doesn’t exist, a FileNotFoundError is raised.

|  |
| --- |
| with open('example.txt', 'r+') as file:      content = file.read()      print(content)      file.write('Appending new content') |

**6.Using the open() function to create and access files**

The Python open() function is used to open internally stored files. It returns the contents of the file as Python objects.

## Python open() Function Syntax

The open() function in [Python](https://www.geeksforgeeks.org/python-programming-language/) has the following syntax:

**Syntax:** open(file\_name, mode)

**Parameters:**

**file\_name:** This parameter as the name suggests, is the name of the file that we want to open.

**mode:** This parameter is a string that is used to specify the mode in which the file is to be opened. The following strings can be used to activate a specific mode:

* **“r”:** This string is used to read(only) the file. It is passed as default if no parameter is supplied and returns an error if no such file exists.
* **“w”:** This string is used for writing on/over the file. If the file with the supplied name doesn’t exist, it creates one for you.
* **“a”:** This string is used to add(append) content to an existing file. If no such file exists, it creates one for you.
* **“x”:** This string is used to create a specific file.
* **“b”:** This string is used when the user wants to handle the file in binary mode. This is generally used to handle image files.
* **“t”:** This string is used to handle files in text mode. By default, the open() function uses the text mode.

In [Python](https://www.geeksforgeeks.org/python-programming-language/), we can open a file by using the open() function already provided to us by Python. By using the open() function, we can open a file in the current directory as well as a file located in a specified location with the help of its path. In this example, we are opening a file “gfg.txt” located in the current directory and “gfg1.txt” located in a specified location.

* Python3

|  |
| --- |
| # opens gfg text file of the current directory  f = open("gfg.txt")    # specifying the full path  f = open("C:/HP/Desktop/gfg1.txt") |

## open() Function in Python Examples

Let us see a few examples of the Python open() function.

### Creating a Text File

The open() function in Python can be used to create a file. Here we will be creating a text file named “geeksforgeeks.txt”.

created\_file = open("geeksforgeeks.txt","x")

# Check the file

print(open("geeksforgeeks.txt","r").read() == False)

**Output:**

False

**7.Closing files using close()**

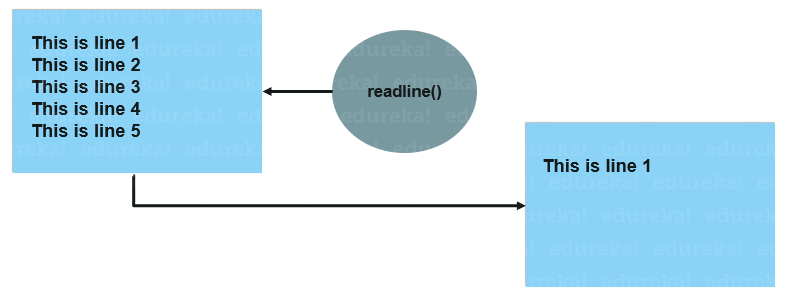
The close() method closes an open file.

You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

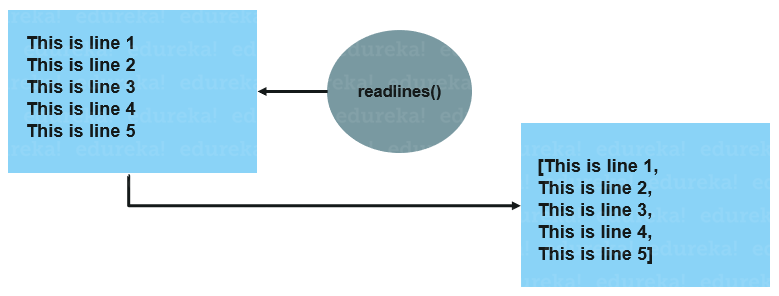
*file*.close()

8.Reading from a files using read(),readline(),readlines()

Python readline() method will return a line from the [file when called](https://www.edureka.co/blog/file-handling-in-python/).



readlines() method will return all the lines in a file in the format of a [list](https://www.edureka.co/blog/lists-in-python/) where each element is a line in the file.



**Python readline() syntax**

file = open("filename.txt", "r")

file.readline()

The readline method takes one parameter i.e size, the default value for the size parameter is -1. It means that the method will return the whole line. It is an optional parameter, we can specify the number of bytes from a line to return.

**readlines() Syntax**

file = open("filename.txt","r")

file.readlines()

The readlines method takes one parameter i.e hint, the default value for the hint parameter is -1. It means that the method will return all the lines. If we specify the hint parameter, the lines exceeding the number of bytes from the hint parameter will not be returned by the readlines method.

**9.Writing to a file using write() and writelines()**

The write() function will write the content in the file without adding any extra characters.

**Syntax**:

# Writes string content referenced by file object.

file\_name.write(content)

As per the syntax, the string that is passed to the write() function is written into the opened file. The string may include numbers, special characters, or symbols. While writing data to a file, we must know that the write function does not add a newline character(\n) to the end of the string. The write() function returns None.

**Example:**

* Python3

|  |
| --- |
| file = open("Employees.txt", "w")    for i in range(3):     name = input("Enter the name of the employee: ")     file.write(name)     file.write("\n")    file.close()    print("Data is written into the file.") |

**Output:**

Data is written into the file.

**Sample Run:**

Enter the name of the employee: Aditya

Enter the name of the employee: Aditi

Enter the name of the employee: Anil

### writelines() function

This function writes the content of a list to a file.

**Syntax**:

# write all the strings present in the list "list\_of\_lines"

# referenced by file object.

file\_name.writelines(list\_of\_lines)

As per the syntax, the list of strings that is passed to the writelines() function is written into the opened file. Similar to the write() function, the writelines() function does not add a newline character(\n) to the end of the string.

**Example:**

* Python3

|  |
| --- |
| file1 = open("Employees.txt", "w")  lst = []  for i in range(3):      name = input("Enter the name of the employee: ")      lst.append(name + '\n')    file1.writelines(lst)  file1.close()  print("Data is written into the file.") |

**Output:**

Data is written into the file.

**Sample Run:**

Enter the name of the employee: Rhea

Enter the name of the employee: Rohan

Enter the name of the employee: Rahul

The only difference between the **write()** and **writelines()** is that write() is used to write a string to an already opened file while writelines() method is used to write a list of strings in an opened file.

**10.Introduction to exceptions and how to handle them using try,except and finally**

An Exception is an Unexpected Event, which occurs during the execution of the program. It is also known as a **run time error**. When that error occurs, [Python](https://www.geeksforgeeks.org/python-programming-language/) generates an exception during the execution and that can be handled, which prevents your program from interrupting.

In this code, The system can not divide the number with zero so an exception is raised.

a = 5

b = 0

print(a/b)

**Output**

Traceback (most recent call last):

File "/home/8a10be6ca075391a8b174e0987a3e7f5.py", line 3, in <module>

print(a/b)

ZeroDivisionError: division by zero

## Exception handling with try, except, else, and finally

* **Try**: This block will test the excepted error to occur
* **Except**:  Here you can handle the error
* **Else**: If there is no exception then this block will be executed
* **Finally**: Finally block always gets executed either exception is generated or not

### Working of ‘try’ and ‘except’

Let’s first understand how the [Python try and except](https://www.geeksforgeeks.org/python-try-except/) works

* First **try** clause is executed i.e. the code between **try** and **except** clause.
* If there is no exception, then only **try** clause will run, **except** clause will not get executed.
* If any exception occurs, the **try** clause will be skipped and **except** clause will run.
* If any exception occurs, but the **except** clause within the code doesn’t handle it, it is passed on to the outer **try** statements. If the exception is left unhandled, then the execution stops.
* A **try** statement can have more than one **except** clause.
* # Python code to illustrate working of try()
* def divide(x, y):
* try:
* # Floor Division : Gives only Fractional
* # Part as Answer
* result = x // y
* print("Yeah ! Your answer is :", result)
* except ZeroDivisionError:
* print("Sorry ! You are dividing by zero ")
* # Look at parameters and note the working of Program
* divide(3, 2)
* divide(3, 0)

O/P:

Yeah ! Your answer is : 1

Sorry ! You are dividing by zero

**11.understanding multiple exceptions and custom exceptions.**

Here’s an example that demonstrates how to use multiple except clauses to handle different exceptions:

try:

x = int(input("Enter a number: "))

result = 10 / x

except ZeroDivisionError:

print("You cannot divide by zero.")

except ValueError:

print("Invalid input. Please enter a valid number.")

except Exception as e:

print(f"An error occurred: {e}")

**Output:**

Enter a number: An error occurred: EOF when reading a line

## Custom Exceptions:

Python also enables you to define your custom exceptions by creating a new class that inherits from the ‘Exception’ class or its subclasses. Custom exceptions can be helpful when you want to provide more specific details about the error or handle specific scenarios differently.

class CustomException(Exception):  
 pass  
  
try:  
 # Code that might raise a custom exception  
except CustomException as e:  
 # Exception handling code for CustomException

In the above code, if a ‘CustomException’ is raised, the corresponding ‘except’ block will handle it.

**12.Understanding the concepts of classes,objects,attributes and methods in python**

A class in Python is a user-defined template for creating objects. It bundles data and functions together, making it easier to manage and use them. When we create a new class, we define a new type of object. We can then create multiple instances of this object type.

Classes are created using **class keyword**. Attributes are variables defined inside the class and represent the properties of the class. Attributes can be accessed using the dot**. operator** (e.g., MyClass.my\_attribute).

## Create a Class

# define a class

class Dog:

sound = "bark" # class attribute

## Create Object

An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

Now, let’s create an object from **Dog class.**

class Dog:

sound = "bark"

# Create an object from the class

dog1 = Dog()

# Access the class attribute

print(dog1.sound)

**sound attribute** is a class attribute. It is shared across all instances of Dog class, so can be directly accessed through instance **dog1**.

**13.Differeent between local and global variables**

**Python Global variables** are those which are not defined inside any function and have a global scope whereas Python **local variables** are those which are defined inside a function and their scope is limited to that function only. In other words, we can say that local variables are accessible only inside the function in which it was initialized whereas the global variables are accessible throughout the program and inside every function.

## Python Local Variables

Local variables in Python are those which are initialized inside a function and belong only to that particular function. It cannot be accessed anywhere outside the function. Let’s see how to create a local variable.

a = 1

# Uses global because there is no local 'a'

def f():

print('Inside f() : ', a)

# Variable 'a' is redefined as a local

def g():

a = 2

print('Inside g() : ', a)

# Uses global keyword to modify global 'a'

def h():

global a

a = 3

print('Inside h() : ', a)

# Global scope

print('global : ', a)

f()

print('global : ', a)

g()

print('global : ', a)

h()

print('global : ', a)

Output

global : 1

Inside f() : 1

global : 1

Inside g() : 2

global : 1

Inside h() : 3

global : 3

**14.Single, multilevel,multiple ,hierarchical and hybrid inheritance in python**

Inheritance is a fundamental concept in [object-oriented programming](https://www.geeksforgeeks.org/python-oops-concepts/)(OOP) that allows a class (called a child or derived class) to inherit attributes and methods from another class (called a parent or base class). This promotes code reuse, modularity, and a hierarchical class structure.

## Types of Python Inheritance

1. **Single Inheritance**: A child class inherits from one parent class.
2. **Multiple Inheritance**: A child class inherits from more than one parent class.
3. **Multilevel Inheritance**: A class is derived from a class which is also derived from another class.
4. **Hierarchical Inheritance**: Multiple classes inherit from a single parent class.
5. **Hybrid Inheritance**: A combination of more than one type of inheritance.

# 1. Single Inheritance

class Person:

def \_\_init\_\_(self, name):

self.name = name

class Employee(Person): # Employee inherits from Person

def \_\_init\_\_(self, name, salary):

super().\_\_init\_\_(name)

self.salary = salary

# 2. Multiple Inheritance

class Job:

def \_\_init\_\_(self, salary):

self.salary = salary

class EmployeePersonJob(Employee, Job): # Inherits from both Employee and Job

def \_\_init\_\_(self, name, salary):

Employee.\_\_init\_\_(self, name, salary) # Initialize Employee

Job.\_\_init\_\_(self, salary) # Initialize Job

# 3. Multilevel Inheritance

class Manager(EmployeePersonJob): # Inherits from EmployeePersonJob

def \_\_init\_\_(self, name, salary, department):

EmployeePersonJob.\_\_init\_\_(self, name, salary) # Explicitly initialize EmployeePersonJob

self.department = department

# 4. Hierarchical Inheritance

class AssistantManager(EmployeePersonJob): # Inherits from EmployeePersonJob

def \_\_init\_\_(self, name, salary, team\_size):

EmployeePersonJob.\_\_init\_\_(self, name, salary) # Explicitly initialize EmployeePersonJob

self.team\_size = team\_size

# 5. Hybrid Inheritance (Multiple + Multilevel)

class SeniorManager(Manager, AssistantManager): # Inherits from both Manager and AssistantManager

def \_\_init\_\_(self, name, salary, department, team\_size):

Manager.\_\_init\_\_(self, name, salary, department) # Initialize Manager

AssistantManager.\_\_init\_\_(self, name, salary, team\_size) # Initialize AssistantManager

# Creating objects to show inheritance

# Single Inheritance

emp = Employee("John", 40000)

print(emp.name, emp.salary)

# Multiple Inheritance

emp2 = EmployeePersonJob("Alice", 50000)

print(emp2.name, emp2.salary)

# Multilevel Inheritance

mgr = Manager("Bob", 60000, "HR")

print(mgr.name, mgr.salary, mgr.department)

# Hierarchical Inheritance

asst\_mgr = AssistantManager("Charlie", 45000, 10)

print(asst\_mgr.name, asst\_mgr.salary, asst\_mgr.team\_size)

# Hybrid Inheritance

sen\_mgr = SeniorManager("David", 70000, "Finance", 20)

print(sen\_mgr.name, sen\_mgr.salary, sen\_mgr.department, sen\_mgr.team\_size)

Output

John 40000

Alice 50000

Bob 60000 HR

Charlie 45000 10

David 70000 Finance 20

**15.Using the super() function to access properties of the parent class**

n [Python](https://www.geeksforgeeks.org/python-programming-language/), the super() function is used to refer to the parent class or superclass. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

class Emp():

def \_\_init\_\_(self, id, name, Add):

self.id = id

self.name = name

self.Add = Add

# Class freelancer inherits EMP

class Freelance(Emp):

def \_\_init\_\_(self, id, name, Add, Emails):

super().\_\_init\_\_(id, name, Add)

self.Emails = Emails

Emp\_1 = Freelance(103, "Suraj kr gupta", "Noida" , "KKK@gmails")

print('The ID is:', Emp\_1.id)

print('The Name is:', Emp\_1.name)

print('The Address is:', Emp\_1.Add)

print('The Emails is:', Emp\_1.Emails)

**Output :**

The ID is: 103  
The Name is: Suraj kr gupta  
The Address is: Noida  
The Emails is: KKK@gmails

**16.Method overloading:defining multiple methods with same name but different parameters.**

**Method Overloading:**

Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

# First product method.

# Takes two argument and print their

# product

def product(a, b):

p = a \* b

print(p)

# Second product method

# Takes three argument and print their

# product

def product(a, b, c):

p = a \* b\*c

print(p)

# Uncommenting the below line shows an error

# product(4, 5)

# This line will call the second product method

product(4, 5, 5)

Output

100

**17.method overriding : redefining a parent class method in the child class**

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, the same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

# Python program to demonstrate

# Defining parent class

class Parent():

# Constructor

def \_\_init\_\_(self):

self.value = "Inside Parent"

# Parent's show method

def show(self):

print(self.value)

# Defining child class

class Child(Parent):

# Constructor

def \_\_init\_\_(self):

super().\_\_init\_\_() # Call parent constructor

self.value = "Inside Child"

# Child's show method

def show(self):

print(self.value)

# Driver's code

obj1 = Parent()

obj2 = Child()

obj1.show() # Should print "Inside Parent"

obj2.show() # Should print "Inside Child"

**Output:**

Inside Parent  
Inside Child

**18.Intoduction to SQLite3 and pyMYSQL for database connectivity**

Connecting to the SQLite Database can be established using the connect() method, passing the name of the database to be accessed as a parameter. If that database does not exist, then it’ll be created.

sqliteConnection = sqlite3.connect('sql.db')

But what if you want to execute some queries after the connection is being made. For that, a cursor has to be created using the cursor() method on the connection instance, which will execute our SQL queries.

cursor = sqliteConnection.cursor()

print('DB Init')

The SQL query to be executed can be written in form of a string, and then executed by calling the execute() method on the cursor object. Then, the result can be fetched from the server by using the fetchall() method, which in this case, is the SQLite Version Number.

query = 'SQL query;'

cursor.execute(query)

result = cursor.fetchall()

print('SQLite Version is {}'.format(result))

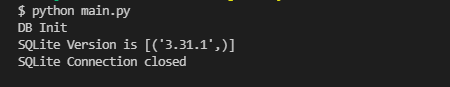
Consider the below example where we will connect to an SQLite database and will run a simple query select sqlite\_version(); to find the version of the SQLite we are using.

Example:

* Python

|  |
| --- |
| import sqlite3    try:        # Connect to DB and create a cursor      sqliteConnection = sqlite3.connect('sql.db')      cursor = sqliteConnection.cursor()      print('DB Init')        # Write a query and execute it with cursor      query = 'select sqlite\_version();'      cursor.execute(query)        # Fetch and output result      result = cursor.fetchall()      print('SQLite Version is {}'.format(result))        # Close the cursor      cursor.close()    # Handle errors  except sqlite3.Error as error:      print('Error occurred - ', error)    # Close DB Connection irrespective of success  # or failure  finally:        if sqliteConnection:          sqliteConnection.close()          print('SQLite Connection closed') |

Output:



The mysql.connector provides the connect() method used to create a connection between the MySQL database and the Python application. The syntax is given below.

Syntax:

Conn\_obj= mysql.connector.connect(host = <hostname>, user = <username>, passwd = <password>)

The connect() function accepts the following arguments.

Hostname – It represents the server name or IP address on which MySQL is running.  
Username – It represents the name of the user that we use to work with the MySQL server. By default, the username for the MySQL database is root.  
Password – The password is provided at the time of installing the MySQL database. We don’t need to pass a password if we are using the root.  
Database – It specifies the database name which we want to connect. This argument is used when we have multiple databases.

In the following example we will be connecting to MySQL database using connect()  
Example:

* Python3

|  |
| --- |
| # Python program to connect  # to mysql database      import mysql.connector      # Connecting from the server  conn = mysql.connector.connect(user = 'username',                                 host = 'localhost',                                database = 'database\_name')    print(conn)    # Disconnecting from the server  conn.close() |

Output:

python-mysql-connect-1

**19.Creating and executing SQL queries from python using these connectors**

For that, we need to install the [MySQL Connector](https://dev.mysql.com/doc/connector-python/en/) Python library. To do this, follow [the instructions](https://dev.mysql.com/doc/connector-python/en/connector-python-installation.html), or just use pip:

pip install mysql-connector-python

### Importing Libraries

As with every project in Python, the very first thing we want to do is import our libraries.

It is best practice to import all the libraries we are going to use at the beginning of the project, so people reading or reviewing our code know roughly what is coming up so there are no surprises.

import mysql.connector

from mysql.connector import Error

## Connecting to MySQL Server

By this point we should have [MySQL Community Server](https://dev.mysql.com/downloads/mysql/) set up on our system. Now we need to write some code in Python that lets us establish a connection to that server.

def create\_server\_connection(host\_name, user\_name, user\_password):

connection = None

try:

connection = mysql.connector.connect(

host=host\_name,

user=user\_name,

passwd=user\_password

)

print("MySQL Database connection successful")

except Error as err:

print(f"Error: '{err}'")

return connection

**20.using re.search() and re.match() functions in python’s re module for pattern matching**

The**re.search()** and **re.match()** both are functions of re module in python. These functions are very efficient and fast for searching in strings. The function searches for some substring in a string and returns a match object if found, else it returns none.

There is a difference between the use of both functions. Both return the first match of a substring found in the string, but **re.match()** searches only from the beginning of the string and return match object if found. But if a match of substring is found somewhere in the middle of the string, it returns none.   
While **re.search()** searches for the whole string even if the string contains multi-lines and tries to find a match of the substring in all the lines of string.

# import re module

import re

Substring ='string'

String1 ='''We are learning regex with geeksforgeeks

regex is very useful for string matching.

It is fast too.'''

String2 ='''string We are learning regex with geeksforgeeks

regex is very useful for string matching.

It is fast too.'''

# Use of re.search() Method

print(re.search(Substring, String1, re.IGNORECASE))

# Use of re.match() Method

print(re.match(Substring, String1, re.IGNORECASE))

# Use of re.search() Method

print(re.search(Substring, String2, re.IGNORECASE))

# Use of re.match() Method

print(re.match(Substring, String2, re.IGNORECASE))

**Output :**

<re.Match object; span=(69, 75), match='string'>  
None  
<re.Match object; span=(0, 6), match='string'>  
<re.Match object; span=(0, 6), match='string'>

**21.Different between search and match.**

String match() and String search() The match() method returns an array of matches. The search() method returns the position of the first match.