**PYTHON LAMBDA FUNCTIONS**

**Python Lambda Functions** are **anonymous functions means that the function is without a name**. As we already know the *def* keyword is used to define a normal function in Python. Similarly, the ***lambda* keyword** is used to **define an anonymous function** in Python.

**Syntax: lambda arguments : expression**

* *This function can have* ***any number of arguments but only one expression****,* ***which is evaluated and returned****.*
* *One is free to use lambda functions wherever* ***function objects*** *are required.*
* *You need to keep in your knowledge that lambda functions are* ***syntactically restricted to a single expression****.*

**Python Lambda Function Example**

In the example, we defined a lambda function(**upper**) to convert a string to its upper case using upper().

This code defines a lambda function named **upper** that takes a string as its argument and converts it to uppercase using the **upper()** method. It then applies this lambda function to the string 'GeeksforGeeks' and prints the result

|  |
| --- |
| str1 **=** 'GeeksforGeeks'  upper **=** **lambda** string: string.upper()  print(upper(str1))  **Output:**  GEEKSFORGEEKS |

|  |
| --- |
| greet = lambda : print('Hello World')  greet()  **Output :**  Hello World |

|  |
| --- |
| greet\_user = lambda name : print('Hey there,', name)  greet\_user('Ram')  **Output :**  Hey there, Ram |

**Difference Between Lambda functions and def defined function**

The code defines a cube function using both the '**def'** keyword and a lambda function. It calculates the cube of a given number (5 in this case) using both approaches and prints the results. The output is 125 for both the '**def'** and lambda functions, demonstrating that they achieve the same cube calculation.

|  |
| --- |
| **def** cube(y):  **return** y**\***y**\***y    lambda\_cube **=** **lambda** y : y**\***y**\***y  print("Using function defined with `def` keyword, cube:", cube(5))  print("Using lambda function, cube:", lambda\_cube(5)) |

**Output:**

Using function defined with 'def' keyword, cube: 125  
Using lambda function, cube: 125

As we can see in the above example, both the **cube()** function and **lambda\_cube()** function behave the same and as intended. Let’s analyze the above example a bit more:

| **WITH LAMBDA FUNCTION** | **WITHOUT LAMBDA FUNCTION** |
| --- | --- |
| Supports **single-line statements** that **return some value**. | Supports **any number of lines** inside a function block |
| Good for performing **short operations/data manipulations**. | Good for **any cases that require multiple lines of code**. |
| Using the lambda function can **sometime reduce the readability of code**. | We can use comments and function descriptions for **easy readability**. |

**Practical Uses of Python lambda function**

**Python Lambda Function with List Comprehension**

On each iteration inside the list comprehension, we are creating a new lambda function with a default argument of x (where x is the current item in the iteration). Later, inside the for loop, we are calling the same function object having the default argument using **item()** and get the desired value. Thus, **is\_even\_list** stores the list of lambda function objects.

|  |
| --- |
| is\_even\_list **=** [**lambda** arg**=**x: arg **\*** 10 **for** x **in** range(1, 5)]  **for** item **in** is\_even\_list:  **print**(item()) |

**Output:**

10  
20  
30  
40

**Python Lambda Function with if-else**

Here we are using the **Max** lambda function to find the maximum of two integers.

|  |
| --- |
| Max **=** **lambda** a, b : a **if**(a > b) **else** b  print(Max(1, 2)) |

**Output:**

2

**Using lambda() Function with filter()**

**The filter() function in Python takes in a function and a list as arguments**. This offers an elegant way to filter out all the elements of a sequence for which the function returns True. Here is a small program that returns the odd numbers from an input list:

**Filter out all odd numbers using filter() and lambda function**

Here, lambda x: (x % 2 != 0) returns True if x is odd. Since filter() only keeps elements where it produces **True**, thus it removes all even numbers that generated **False**.

|  |
| --- |
| li **=** [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]  final\_list **=** list(filter(**lambda** x: (x **%** 2 !**=** 0), li))  print(final\_list) |

**Output:**

[5, 7, 97, 77, 23, 73, 61]

**Filter all people having age more than 18, using lambda and filter() function**

The code filters a list of ages and extracts the ages of adults (ages greater than 18) using a lambda function and the **‘filter'** function. It then prints the list of adult ages. The output displays the ages of individuals who are 18 years or older.

|  |
| --- |
| ages **=** [13, 90, 17, 59, 21, 60, 5]  adults **=** list(filter(**lambda** age: age > 18, ages))    print(adults) |

**Output:**

[90, 59, 21, 60]

**Using lambda() Function with map()**

**The map() function in Python takes in a function and a list as an argument**. The function is called with a lambda function and a list and a **new list is returned** which contains all the **lambda-modified items returned** **by that function for each item.** Example:

**Multiply all elements of a list by 2 using lambda and map() function**

The code doubles each element in a list using a lambda function and the '**map**' function. It then prints the new list with the doubled elements. The output displays each element from the original list, multiplied by 2.

|  |
| --- |
| li **=** [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]    final\_list **=** list(map(**lambda** x: x**\***2, li))  print(final\_list) |

**Output:**

[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]

**Transform all elements of a list to upper case using lambda and map() function**

The code converts a list of animal names to uppercase using a lambda function and the **‘map'** function. It then prints the list with the animal names in uppercase. The output displays the animal names in all uppercase letters.

|  |
| --- |
| animals **=** ['dog', 'cat', 'parrot', 'rabbit']  uppered\_animals **=** list(map(**lambda** animal: animal.upper(), animals))    print(uppered\_animals) |

**Output:**

['DOG', 'CAT', 'PARROT', 'RABBIT']

**Using lambda() Function with reduce()**

The reduce() function in Python **takes in a function and a list as an argument**. The function is called with a lambda function and an iterable and a **new reduced result is returned**. This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the ***functools*module**.

**A sum of all elements in a list using lambda and reduce() function**

The code calculates the sum of elements in a list using the '**reduce'** function from the **‘functools'** module. It imports '**reduce'**, defines a list, applies a lambda function that adds two elements at a time, and prints the sum of all elements in the list. The output displays the computed sum.

|  |
| --- |
| **from** functools **import** reduce  li **=** [5, 8, 10, 20, 50, 100]  sum **=** reduce((**lambda** x, y: x **+** y), li)  print(sum) |

**Output:**

193

Here the results of the previous two elements are added to the next element and this goes on till the end of the list like (((((5+8)+10)+20)+50)+100).

**Find the maximum element in a list using lambda and reduce() function**

The code uses the '**functools**' module to find the maximum element in a list ('**list**') by employing the '**reduce'** function and a lambda function. It then prints the maximum element, which is 6 in this case.

|  |
| --- |
| **import** functools  list **=** [1, 3, 5, 6, 2]  **print**("The maximum element of the list is : ", end**=**"")  print(functools.reduce(**lambda** a, b: a **if** a > b **else** b, list)) |

**Output:**

The maximum element of the list is : 6

**UNBOUND VARIABLES**

In Python, **Unbound Variables** refer to variables that are **used in a scope without being defined or assigned a value within that scope**. When you try to use a variable before assigning a value to it, Python raises a **NameError** indicating that the variable is not defined.

Here's an example of using an unbound variable:

def example\_function():

print(x) # Trying to use variable x without defining it

example\_function()

In this example, the variable **x** is not defined within the **example\_function**, so trying to print it will result in a **NameError**. To avoid this error, you should define or assign a value to the variable before using it:

def example\_function():

x = 10 # Define and assign a value to the variable x

print(x)

example\_function()

In this corrected example, the variable **x** is **defined and assigned a value before being used**, **preventing any unbound variable issues**. It's essential to ensure that variables are properly defined within the scope where they are used to avoid such errors.

**EXCEPTION HANDLING**

In Python, an **Exception** is an **unexpected event or error** that **disrupts** the **normal flow of a program during its execution.**

When the **interpreter** **encounters a situation it cannot handle**, such as attempting an invalid operation or accessing non-existent data, it **raises an exception**.

Exception handling **enhances code maintainability and user experience** by providing a **structured way to deal with errors and unexpected situations**.

The **primary components** of exception handling in Python include the **try**, **except**, **else**, and **finally** blocks.

**Try**:

The **try** block **encapsulates the code where an exception might occur**. It is the section where **potential errors are anticipated**

**Except:**

The except block is executed **if an exception occurs in the corresponding try block**. It **catches and handles the exception**. Specific exceptions can be caught, or a more general except block can handle any exception.

**Else:**

The else block is **executed** **if no exceptions are raised in the try block**. It allows for the execution of code that should only run when no errors occur.

**Finally:**

The finally block is **always executed**, regardless of **whether an exception occurs or not**. It is often used for **cleanup operations**, such as **closing files or releasing resources**.

**Syntax**

1. **try**:
2. # Code block
3. # These statements are those which can probably have some error
5. **except**:
6. # This block is optional.
7. # If the try block encounters an exception, this block will handle it.
9. **else**:
10. # If there is no exception, this code block will be executed by the Python interpreter
12. **finally**:
13. # Python interpreter will always execute this code.

**Example 1 :**

try:

result = 10 / 0 # This will raise a ZeroDivisionError

except ZeroDivisionError:

print("Cannot divide by zero!")

else:

print("Division successful!")

finally:

print("This will always be executed, regardless of exceptions.")

**Output :**

Cannot divide by zero!

This will always be executed, regardless of exceptions.

**Example 2 :**

def get\_user\_input():

try:

user\_input = int(input("Enter a number : "))

print("You entered :", user\_input)

except ValueError:

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

# Main program

get\_user\_input()

**Output 1 :**

Enter a number : 45

You entered : 45

**Output 2 :**

Enter a number : Raj

Invalid input. Please enter a valid number.

# Python Modules

**Python Module** is a file that contains built-in functions, classes,**its** and variables. There are many **Python modules**, each with its specific work.

In this article, we will cover all about Python modules, such as How to create our own simple module, Import Python modules, From statements in Python, we can use the alias to rename the module, etc.

## What is Python Module

A [Python](https://www.geeksforgeeks.org/python-programming-language/)module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

## ****Create a Python Module****

To create a Python module, write the desired code and save that in a file with**.py**extension. Let’s understand it better with an example:

**Example:**

Let’s create a simple calc.py in which we define two functions, one **add** and another **subtract**.

|  |
| --- |
| # A simple module, calc.py  **def** add(x, y):  **return** (x**+**y)    **def** subtract(x, y):  **return** (x**-**y) |

## ****Import module in Python****

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

***Note***: A search path is a list of directories that the interpreter searches for importing a module.

For example, to import the module calc.py, we need to put the following command at the top of the script.

### ****Syntax to Import Module in Python****

import module

**Note:**This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

**Importing modules in Python Example**

Now, we are importing the **calc** that we created earlier to perform add operation.

|  |
| --- |
| # importing  module calc.py  **import** calc    print(calc.add(10, 2)) |

**Output:**

12

## ****Python Import From Module****

Python’s from statement lets you import specific attributes from a module without importing the module as a whole.

### Import Specific Attributes from a Python module

Here, we are importing specific sqrt and factorial attributes from the math module.

|  |
| --- |
| # importing sqrt() and factorial from the  # module math  **from** math **import** sqrt, factorial    # if we simply do "import math", then  # math.sqrt(16) and math.factorial()  # are required.  **print**(sqrt(16))  **print**(factorial(6)) |

**Output:**

4.0  
720

## Import all Names

The \* symbol used with the import statement is used to import all the names from a module to a current namespace.

**Syntax:**

from module\_name import \*

### What does import \* do in Python?

The use of \* has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use \*, else do so.

|  |
| --- |
| # importing sqrt() and factorial from the  # module math  **from** math **import** **\***    # if we simply do "import math", then  # math.sqrt(16) and math.factorial()  # are required.  print(sqrt(16))  print(factorial(6)) |

**Output**

4.0  
720

## Locating Python Modules

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the [built-in module](https://www.geeksforgeeks.org/built-in-modules-in-python/), if not found then it looks for a list of directories defined in the [sys.path](https://www.geeksforgeeks.org/sys-path-in-python/). Python interpreter searches for the module in the following manner –

* First, it searches for the module in the current directory.
* If the module isn’t found in the current directory, Python then searches each directory in the shell variable [PYTHONPATH](https://www.geeksforgeeks.org/pythonpath-environment-variable-in-python/). The PYTHONPATH is an environment variable, consisting of a list of directories.
* If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

### ****Directories List for Modules****

Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

## Python3

|  |
| --- |
| # importing sys module  **import** sys    # importing sys.path  print(sys.path) |

**Output:**

*[‘/home/nikhil/Desktop/gfg’, ‘/usr/lib/python38.zip’, ‘/usr/lib/python3.8’, ‘/usr/lib/python3.8/lib-dynload’, ”, ‘/home/nikhil/.local/lib/python3.8/site-packages’, ‘/usr/local/lib/python3.8/dist-packages’, ‘/usr/lib/python3/dist-packages’, ‘/usr/local/lib/python3.8/dist-packages/IPython/extensions’, ‘/home/nikhil/.ipython’]*

## Renaming the Python Module

We can rename the module while importing it using the keyword.

***Syntax:****Import****Module\_name****as****Alias\_name***

## Python3

|  |
| --- |
| # importing sqrt() and factorial from the  # module math  **import** math as mt    # if we simply do "import math", then  # math.sqrt(16) and math.factorial()  # are required.  **print**(mt.sqrt(16))  print(mt.factorial(6)) |

**Output**

4.0

720

## ****Python Built-in modules****

There are several built-in modules in Python, which you can import whenever you like.

|  |
| --- |
| # importing built-in module math  **import** math    # using square root(sqrt) function contained  # in math module  **print**(math.sqrt(25))    # using pi function contained in math module  **print**(math.pi)    # 2 radians = 114.59 degrees  print(math.degrees(2))    # 60 degrees = 1.04 radians  print(math.radians(60))    # Sine of 2 radians  **print**(math.sin(2))    # Cosine of 0.5 radians  **print**(math.cos(0.5))    # Tangent of 0.23 radians  print(math.tan(0.23))    # 1 \* 2 \* 3 \* 4 = 24  **print**(math.factorial(4))    # importing built in module random  **import** random    # printing random integer between 0 and 5  **print**(random.randint(0, 5))    # print random floating point number between 0 and 1  print(random.random())    # random number between 0 and 100  print(random.random() **\*** 100)    List **=** [1, 4, True, 800, "python", 27, "hello"]    # using choice function in random module for choosing  # a random element from a set such as a list  **print**(random.choice(List))      # importing built in module datetime  **import** datetime  **from** datetime **import** date  **import** time    # Returns the number of seconds since the  # Unix Epoch, January 1st 1970  print(time.time())    # Converts a number of seconds to a date object  **print**(date.fromtimestamp(454554)) |

**Output:**

5.0  
3.14159265359  
114.591559026  
1.0471975512  
0.909297426826  
0.87758256189  
0.234143362351  
24  
3  
0.401533172951  
88.4917616788  
True  
1461425771.87

# Python String

A String is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it. Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

## ****What is a String in Python?****

[Python](https://www.geeksforgeeks.org/python-programming-language/) does not have a character data type, a single character is simply a string with a length of 1.

**Example:**

"Geeksforgeeks" or 'Geeksforgeeks' or "a"

## Python3

|  |
| --- |
| print("A Computer Science portal for geeks")  print('A') |

**Output:**

A Computer Science portal for geeks  
A

## Creating a String in Python

**Strings in Python** can be created using single quotes or double quotes or even triple quotes. Let us see how we can define a string in Python.

**Example:**

In this example, we will demonstrate different ways to create a Python String. We will create a string using single quotes (‘ ‘), double quotes (” “), and triple double quotes (“”” “””). The triple quotes can be used to declare multiline strings in Python.

## Python3

|  |
| --- |
| # Python Program for  # Creation of 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)    # Creating a String  # with triple Quotes  String1 **=** '''I'm a Geek and I live in a world of "Geeks"'''  print("\nString with the use of Triple Quotes: ")  print(String1)    # Creating String with triple  # Quotes allows multiple lines  String1 **=** '''Geeks              For              Life'''  print("\nCreating a multiline String: ")  print(String1) |

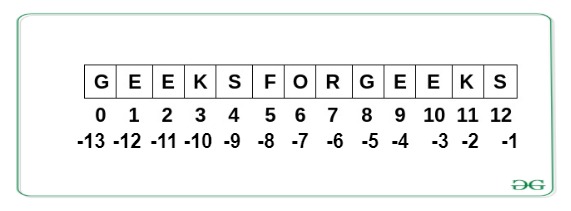
**Output:**

String with the use of Single Quotes:   
Welcome to the Geeks World  
String with the use of Double Quotes:   
I'm a Geek  
String with the use of Triple Quotes:   
I'm a Geek and I live in a world of "Geeks"  
Creating a multiline String:   
Geeks  
 For  
 Life

## Accessing characters in Python 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.

While accessing an index out of the range will cause an **IndexError**. Only Integers are allowed to be passed as an index, float or other types that will cause a **TypeError**.



*Python String indexing*

**Example:**

In this example, we will define a string in Python and access its characters using positive and negative indexing. The 0th element will be the first character of the string whereas the -1th element is the last character of the string.

## Python3

|  |
| --- |
| # Python Program to Access  # characters of String    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 cha racter of String is:   
s

## String Slicing

In Python, the [String Slicing](https://www.geeksforgeeks.org/string-slicing-in-python/) method is used to access a range of characters in the String. Slicing in a String is done by using a Slicing operator, i.e., a colon (:).  One thing to keep in mind while using this method is that the string returned after slicing includes the character at the start index but not the character at the last index.

**Example:**

In this example, we will use the string-slicing method to extract a substring of the original string. The [3:12] indicates that the string slicing will start from the 3rd index of the string to the 12th index, (12th character not including). We can also use negative indexing in string slicing.

## Python3

|  |
| --- |
| # Python Program to  # demonstrate String slicing    # Creating a String  String1 **=** "GeeksForGeeks"  print("Initial String: ")  **print**(String1)    # Printing 3rd to 12th character  print("\nSlicing characters from 3-12: ")  **print**(String1[3:12])    # Printing characters between  # 3rd and 2nd last character  print("\nSlicing characters between " **+**        "3rd and 2nd last character: ")  print(String1[3:**-**2]) |

**Output:**

Initial String:   
GeeksForGeeks  
Slicing characters from 3-12:   
ksForGeek  
Slicing characters between 3rd and 2nd last character:   
ksForGee

## Reversing a Python String

By accessing characters from a string, we can also [reverse strings in Python](https://www.geeksforgeeks.org/reverse-string-python-5-different-ways/). We can Reverse a string by using String slicing method.

**Example:**

In this example, we will reverse a string by accessing the index. We did not specify the first two parts of the slice indicating that we are considering the whole string, from the start index to the last index.

## Python3

|  |
| --- |
| #Program to reverse a string  gfg **=** "geeksforgeeks"  print(gfg[::**-**1]) |

**Output:**

skeegrofskeeg

**Example:**

We can also reverse a string by using built-in [join](https://www.geeksforgeeks.org/python-string-join-method/) and [reversed](https://www.geeksforgeeks.org/python-reversed-function/) functions, and passing the string as the parameter to the reversed() function.

## Python3

|  |
| --- |
| # Program to reverse a string    gfg **=** "geeksforgeeks"    # Reverse the string using reversed and join function  gfg **=** "".join(reversed(gfg))    print(gfg) |

**Output:**

skeegrofskeeg

## Deleting/Updating from a String

In Python, the Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. Although deletion of the entire String is possible with the use of a built-in del keyword. This is because Strings are immutable, hence elements of a String cannot be changed once assigned. Only new strings can be reassigned to the same name.

### Updating a character

A character of a string can be updated in Python by first converting the string into a [Python List](https://www.geeksforgeeks.org/python-lists/) and then updating the element in the list. As lists are mutable in nature, we can update the character and then convert the list back into the String.

Another method is using the string slicing method. Slice the string before the character you want to update, then add the new character and finally add the other part of the string again by string slicing.

**Example:**

In this example, we are using both the list and the string slicing method to update a character. We converted the String1 to a list, changes its value at a particular element, and then converted it back to a string using the Python [string join()](https://www.geeksforgeeks.org/python-string-join-method/) method.

In the string-slicing method, we sliced the string up to the character we want to update, concatenated the new character, and finally concatenate the remaining part of the string.

## Python3

|  |
| --- |
| # Python Program to Update  # character of a String    String1 **=** "Hello, I'm a Geek"  **print**("Initial String: ")  print(String1)    # Updating a character of the String  ## As python strings are immutable, they don't support item updation directly  ### there are following two ways  #1  list1 **=** list(String1)  list1[2] **=** 'p'  String2 **=** ''.join(list1)  **print**("\nUpdating character at 2nd Index: ")  print(String2)    #2  String3 **=** String1[0:2] **+** 'p' **+** String1[3:]  print(String3) |

**Output:**

Initial String:   
Hello, I'm a Geek  
Updating character at 2nd Index:   
Heplo, I'm a Geek  
Heplo, I'm a Geek

### Updating Entire String

As Python strings are immutable in nature, we cannot update the existing string. We can only assign a completely new value to the variable with the same name.

**Example:**

In this example, we first assign a value to ‘String1’ and then updated it by assigning a completely different value to it. We simply changed its reference.

## Python3

|  |
| --- |
| # Python Program to Update  # entire String    String1 **=** "Hello, I'm a Geek"  **print**("Initial String: ")  **print**(String1)    # Updating a String  String1 **=** "Welcome to the Geek World"  print("\nUpdated String: ")  print(String1) |

**Output:**

Initial String:   
Hello, I'm a Geek  
Updated String:   
Welcome to the Geek World

### Deleting a character

Python strings are immutable, that means we cannot delete a character from it. When we try to delete thecharacter using the **del** keyword, it will generate an error.

## Python3

|  |
| --- |
| # Python Program to delete  # character of a String    String1 **=** "Hello, I'm a Geek"  print("Initial String: ")  print(String1)    print("Deleting character at 2nd Index: ")  **del** String1[2]  **print**(String1) |

**Output:**

Initial String:   
Hello, I'm a Geek  
Deleting character at 2nd Index:   
Traceback (most recent call last):  
 File "e:\GFG\Python codes\Codes\demo.py", line 9, in <module>  
 del String1[2]  
TypeError: 'str' object doesn't support item deletion

But using slicing we can remove the character from the original string and store the result in a new string.

**Example:**

In this example, we will first slice the string up to the character that we want to delete and then concatenate the remaining string next from the deleted character.

## Python3

|  |
| --- |
| # Python Program to Delete  # characters from a String    String1 **=** "Hello, I'm a Geek"  **print**("Initial String: ")  **print**(String1)    # Deleting a character  # of the String  String2 **=** String1[0:2] **+** String1[3:]  print("\nDeleting character at 2nd Index: ")  print(String2) |

**Output:**

Initial String:   
Hello, I'm a Geek  
Deleting character at 2nd Index:   
Helo, I'm a Geek

### Deleting Entire String

Deletion of the entire string is possible with the use of del keyword. Further, if we try to print the string, this will produce an error because the String is deleted and is unavailable to be printed.

## Python3

|  |
| --- |
| # Python Program to Delete  # entire String    String1 **=** "Hello, I'm a Geek"  **print**("Initial String: ")  print(String1)    # Deleting a String  # with the use of del  **del** String1  print("\nDeleting entire String: ")  print(String1) |

**Error:**

Traceback (most recent call last):   
File "/home/e4b8f2170f140da99d2fe57d9d8c6a94.py", line 12, in   
print(String1)   
NameError: name 'String1' is not defined

## Escape Sequencing in Python

While printing Strings with single and double quotes in it causes **SyntaxError** because String already contains Single and Double Quotes and hence cannot be printed with the use of either of these. Hence, to print such a String either Triple Quotes are used or Escape sequences are used to print Strings.

Escape sequences start with a backslash and can be interpreted differently. If single quotes are used to represent a string, then all the single quotes present in the string must be escaped and the same is done for Double Quotes.

**Example:**

## Python3

|  |
| --- |
| # Python Program for  # Escape Sequencing  # of String    # Initial String  String1 **=** '''I'm a "Geek"'''  **print**("Initial String with use of Triple Quotes: ")  print(String1)    # Escaping Single Quote  String1 **=** 'I\'m a "Geek"'  **print**("\nEscaping Single Quote: ")  print(String1)    # Escaping Double Quotes  String1 **=** "I'm a \"Geek\""  **print**("\nEscaping Double Quotes: ")  print(String1)    # Printing Paths with the  # use of Escape Sequences  String1 **=** "C:\\Python\\Geeks\\"  print("\nEscaping Backslashes: ")  print(String1)    # Printing Paths with the  # use of Tab  String1 **=** "Hi\tGeeks"  print("\nTab: ")  print(String1)    # Printing Paths with the  # use of New Line  String1 **=** "Python\nGeeks"  **print**("\nNew Line: ")  print(String1) |

**Output:**

Initial String with use of Triple Quotes:   
I'm a "Geek"  
Escaping Single Quote:   
I'm a "Geek"  
Escaping Double Quotes:   
I'm a "Geek"  
Escaping Backslashes:   
C:\Python\Geeks\  
Tab:   
Hi Geeks  
New Line:   
Python  
Geeks

**Example:**

To ignore the escape sequences in a String, **r** or **R** is used, this implies that the string is a raw string and escape sequences inside it are to be ignored.

## Python3

|  |
| --- |
| # Printing hello in octal  String1 **=** "\110\145\154\154\157"  print("\nPrinting in Octal with the use of Escape Sequences: ")  print(String1)    # Using raw String to  # ignore Escape Sequences  String1 **=** r"This is \110\145\154\154\157"  **print**("\nPrinting Raw String in Octal Format: ")  **print**(String1)    # Printing Geeks in HEX  String1 **=** "This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"  print("\nPrinting in HEX with the use of Escape Sequences: ")  **print**(String1)    # Using raw String to  # ignore Escape Sequences  String1 **=** r"This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"  print("\nPrinting Raw String in HEX Format: ")  print(String1) |

**Output:**

Printing in Octal with the use of Escape Sequences:   
Hello  
Printing Raw String in Octal Format:   
This is \110\145\154\154\157  
Printing in HEX with the use of Escape Sequences:   
This is Geeks in HEX  
Printing Raw String in HEX Format:   
This is \x47\x65\x65\x6b\x73 in \x48\x45\x58

## Formatting of Strings

Strings in Python can be formatted with the use of [format()](https://www.geeksforgeeks.org/python-string-format-method/) method which is a very versatile and powerful tool for formatting Strings. Format method in String contains curly braces {} as placeholders which can hold arguments according to position or keyword to specify the order.

**Example 1:**

In this example, we will declare a string which contains the curly braces {} that acts as a placeholders and provide them values to see how string declaration position matters.

## Python3

|  |
| --- |
| # Python Program for  # Formatting of Strings    # Default order  String1 **=** "{} {} {}".format('Geeks', 'For', 'Life')  print("Print String in default order: ")  print(String1)    # Positional Formatting  String1 **=** "{1} {0} {2}".format('Geeks', 'For', 'Life')  **print**("\nPrint String in Positional order: ")  **print**(String1)    # Keyword Formatting  String1 **=** "{l} {f} {g}".format(g**=**'Geeks', f**=**'For', l**=**'Life')  print("\nPrint String in order of Keywords: ")  print(String1) |

**Output:**

Print String in default order:   
Geeks For Life  
Print String in Positional order:   
For Geeks Life  
Print String in order of Keywords:   
Life For Geeks

**Example 2:**

Integers such as Binary, hexadecimal, etc., and floats can be rounded or displayed in the exponent form with the use of format specifiers.

## Python3

|  |
| --- |
| # Formatting of Integers  String1 **=** "{0:b}".format(16)  **print**("\nBinary representation of 16 is ")  print(String1)    # Formatting of Floats  String1 **=** "{0:e}".format(165.6458)  print("\nExponent representation of 165.6458 is ")  print(String1)    # Rounding off Integers  String1 **=** "{0:.2f}".format(1**/**6)  print("\none-sixth is : ")  print(String1) |

**Output:**

Binary representation of 16 is   
10000  
Exponent representation of 165.6458 is   
1.656458e+02  
one-sixth is :   
0.17

**Example 3:**

A string can be left, right, or center aligned with the use of format specifiers, separated by a colon(:). The (<) indicates that the string should be aligned to the left, (>) indicates that the string should be aligned to the right and (^) indicates that the string should be aligned to the center. We can also specify the length in which it should be aligned. For example, (<10) means that the string should be aligned to the left within a field of width of 10 characters.

## Python3

|  |
| --- |
| # String alignment  String1 **=** "|{:<10}|{:^10}|{:>10}|".format('Geeks',                                            'for',                                            'Geeks')  print("\nLeft, center and right alignment with Formatting: ")  **print**(String1)    # To demonstrate aligning of spaces  String1 **=** "\n{0:^16} was founded in {1:<4}!".format("GeeksforGeeks",                                                      2009)  print(String1) |

**Output:**

Left, center and right alignment with Formatting:   
|Geeks | for | Geeks|  
 GeeksforGeeks was founded in 2009 !

**Example 4:**

Old-style formatting was done without the use of the format method by usingthe**%**operator

## Python3

|  |
| --- |
| # Python Program for  # Old Style Formatting  # of Integers    Integer1 **=** 12.3456789  print("Formatting in 3.2f format: ")  print('The value of Integer1 is %3.2f' **%** Integer1)  print("\nFormatting in 3.4f format: ")  print('The value of Integer1 is %3.4f' **%** Integer1) |

**Output:**

Formatting in 3.2f format:   
The value of Integer1 is 12.35  
Formatting in 3.4f format:   
The value of Integer1 is 12.3457

# Python String Methods

***Python string methods***is a collection of in-built Python functions that operates on lists.

***Note:****Every string method does not change the original string instead returns a new string with the changed attributes.*

[**Python string**](https://www.geeksforgeeks.org/python-strings/) is a sequence of Unicode characters that is enclosed in quotation marks. In this article, we will discuss the in-built string functions i.e. the functions provided by Python to operate on strings.

## Case Changing of Strings

The below Python functions are used to change the case of the strings. Let’s look at some Python string methods with examples:

* [**lower():**](https://www.geeksforgeeks.org/python-string-lower/) Converts all uppercase characters in a string into lowercase
* [**upper():**](https://www.geeksforgeeks.org/python-string-upper/) Converts all lowercase characters in a string into uppercase
* [**title():**](https://www.geeksforgeeks.org/title-in-python/) Convert string to title case
* [**swapcase():**](https://www.geeksforgeeks.org/python-string-swapcase/)Swap the cases of all characters in a string
* [**capitalize()**](https://www.geeksforgeeks.org/string-capitalize-python/)**:** Convert the first character of a string to uppercase

**Example:**Changing the case of Python Strings

|  |
| --- |
| # Python3 program to show the  # working of upper() function  text **=** 'geeKs For geEkS'    # upper() function to convert  # string to upper case  print("\nConverted String:")  print(text.upper())    # lower() function to convert  # string to lower case  **print**("\nConverted String:")  **print**(text.lower())    # converts the first character to  # upper case and rest to lower case  print("\nConverted String:")  **print**(text.title())    #swaps the case of all characters in the string  # upper case character to lowercase and viceversa  print("\nConverted String:")  **print**(text.swapcase())    # convert the first character of a string to uppercase  **print**("\nConverted String:")  **print**(text.capitalize())    # original string never changes  **print**("\nOriginal String")  print(text) |

**Output**

Converted String:

GEEKS FOR GEEKS

Converted String:

geeks for geeks

Converted String:

Geeks For Geeks

Converted String:

GEEkS fOR GEeKs

Original String

geeKs For geEkS

## List of Python String Methods

Here is the list of in-built Python string methods, that you can use to perform actions on string:

| **Function Name** | **Description** |
| --- | --- |
| [capitalize()](https://www.geeksforgeeks.org/string-capitalize-python/) | Converts the first character of the string to a capital (uppercase) letter |
| [casefold()](https://www.geeksforgeeks.org/casefold-string-python/) | Implements caseless string matching |
| [center()](https://www.geeksforgeeks.org/string-center-python/) | Pad the string with the specified character. |
| [count()](https://www.geeksforgeeks.org/python-string-count/) | Returns the number of occurrences of a substring in the string. |
| [encode()](https://www.geeksforgeeks.org/python-strings-encode-method/) | Encodes strings with the specified encoded scheme |
| [endswith()](https://www.geeksforgeeks.org/string-endswith-python/) | Returns “True” if a string ends with the given suffix |
| [expandtabs()](https://www.geeksforgeeks.org/python-expandtabs-method/) | Specifies the amount of space to be substituted with the “\t” symbol in the string |
| [find()](https://www.geeksforgeeks.org/python-string-find/) | Returns the lowest index of the substring if it is found |
| [format()](https://www.geeksforgeeks.org/python-format-function/) | Formats the string for printing it to console |
| [format\_map()](https://www.geeksforgeeks.org/python-string-format_map/) | Formats specified values in a string using a dictionary |
| [index()](https://www.geeksforgeeks.org/python-string-index-applications/) | Returns the position of the first occurrence of a substring in a string |
| [isalnum()](https://www.geeksforgeeks.org/python-string-isalnum/) | Checks whether all the characters in a given string is alphanumeric or not |
| [isalpha()](https://www.geeksforgeeks.org/python-string-isalpha-application/) | Returns “True” if all characters in the string are alphabets |
| [isdecimal()](https://www.geeksforgeeks.org/python-string-isdecimal/) | Returns true if all characters in a string are decimal |
| [isdigit()](https://www.geeksforgeeks.org/python-string-isdigit-application/) | Returns “True” if all characters in the string are digits |
| [isidentifier()](https://www.geeksforgeeks.org/python-string-isidentifier/) | Check whether a string is a valid identifier or not |
| [islower()](https://www.geeksforgeeks.org/python-string-islower-method/) | Checks if all characters in the string are lowercase |
| [isnumeric()](https://www.geeksforgeeks.org/python-string-isnumeric-application/) | Returns “True” if all characters in the string are numeric characters |
| [isprintable()](https://www.geeksforgeeks.org/isprintable-python-application/) | Returns “True” if all characters in the string are printable or the string is empty |
| [isspace()](https://www.geeksforgeeks.org/python-string-isspace-application/) | Returns “True” if all characters in the string are whitespace characters |
| [istitle()](https://www.geeksforgeeks.org/python-string-istitle/) | Returns “True” if the string is a title cased string |
| [isupper()](https://www.geeksforgeeks.org/python-string-isupper-method/) | Checks if all characters in the string are uppercase |
| [join()](https://www.geeksforgeeks.org/join-function-python/) | Returns a concatenated String |
| [ljust](https://www.geeksforgeeks.org/string-rjust-ljust-python/)() | Left aligns the string according to the width specified |
| [lower()](https://www.geeksforgeeks.org/python-string-lower/) | Converts all uppercase characters in a string into lowercase |
| [lstrip()](https://www.geeksforgeeks.org/python-string-lstrip-method/) | Returns the string with leading characters removed |
| [maketrans](https://www.geeksforgeeks.org/python-maketrans-translate-functions/)() | Returns a translation table |
| [partition()](https://www.geeksforgeeks.org/string-partition-python/) | Splits the string at the first occurrence of the separator |
| [replace()](https://www.geeksforgeeks.org/python-string-replace/) | Replaces all occurrences of a substring with another substring |
| [rfind()](https://www.geeksforgeeks.org/python-string-rfind/) | Returns the highest index of the substring |
| [rindex()](https://www.geeksforgeeks.org/string-rindex-python/) | Returns the highest index of the substring inside the string |
| [rjust()](https://www.geeksforgeeks.org/string-rjust-ljust-python/) | Right aligns the string according to the width specified |
| [rpartition()](https://www.geeksforgeeks.org/python-string-rpartition/) | Split the given string into three parts |
| [rsplit()](https://www.geeksforgeeks.org/python-string-rsplit/) | Split the string from the right by the specified separator |
| [rstrip()](https://www.geeksforgeeks.org/python-string-rstrip/) | Removes trailing characters |
| [splitlines()](https://www.geeksforgeeks.org/python-string-splitlines/) | Split the lines at line boundaries |
| [startswith()](https://www.geeksforgeeks.org/python-string-startswith/) | Returns “True” if a string starts with the given prefix |
| [strip()](https://www.geeksforgeeks.org/python-string-strip/) | Returns the string with both leading and trailing characters |
| [swapcase()](https://www.geeksforgeeks.org/python-string-swapcase/) | Converts all uppercase characters to lowercase and vice versa |
| [title()](https://www.geeksforgeeks.org/title-in-python/) | Convert string to title case |
| [translate()](https://www.geeksforgeeks.org/python-string-translate/) | Modify string according to given translation mappings |
| [upper()](https://www.geeksforgeeks.org/python-string-upper/) | Converts all lowercase characters in a string into uppercase |
| [zfill()](https://www.geeksforgeeks.org/python-string-zfill/) | Returns a copy of the string with ‘0’ characters padded to the left side of the string |

# Regular Expression (RegEx) in Python with Examples

A **Regular Expression or RegEx**is a special sequence of characters that uses a search pattern to find a string or set of strings.

It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.

## Regex Module in Python

[Python](https://www.geeksforgeeks.org/python-programming-language/)has a built-in module named “**re”**that is used for regular expressions in Python. We can import this module by using the [import statement](https://www.geeksforgeeks.org/import-module-python/).

**Example:** Importing re module in Python

* Python3

|  |
| --- |
| # importing re module  **import** re |

## How to Use RegEx in Python?

You can use RegEx in Python after importing re module.

### ****Example:****

This Python code uses regular expressions to search for the word **“portal”**in the given string and then prints the start and end indices of the matched word within the string.

* Python3

|  |
| --- |
| **import** re    s **=** 'GeeksforGeeks: A computer science portal for geeks'    match **=** re.search(r'portal', s)    **print**('Start Index:', match.start())  print('End Index:', match.end()) |

**Output**

Start Index: 34

End Index: 40

**Note:**Here r character (r’portal’) stands for raw, not regex. The raw string is slightly different from a regular string, it won’t interpret the \ character as an escape character. This is because the regular expression engine uses \ character for its own escaping purpose.

Before starting with the Python regex module let’s see how to actually write regex using metacharacters or special sequences.

## Metacharacters

Metacharacters are the characters with special meaning.

To understand the RE analogy, Metacharacters are useful and important. They will be used in functions of module re. Below is the list of metacharacters.

| **MetaCharacters** | **Description** |
| --- | --- |
| \ | Used to drop the special meaning of character following it |
| [] | Represent a character class |
| ^ | Matches the beginning |
| $ | Matches the end |
| . | Matches any character except newline |
| | | Means OR (Matches with any of the characters separated by it. |
| ? | Matches zero or one occurrence |
| \* | Any number of occurrences (including 0 occurrences) |
| + | One or more occurrences |
| {} | Indicate the number of occurrences of a preceding regex to match. |
| () | Enclose a group of Regex |

Let’s discuss each of these metacharacters in detail:

### 1. \ – Backslash

The backslash (\) makes sure that the character is not treated in a special way. This can be considered a way of escaping metacharacters.

For example, if you want to search for the dot(.) in the string then you will find that dot(.) will be treated as a special character as is one of the metacharacters (as shown in the above table). So for this case, we will use the backslash(\) just before the dot(.) so that it will lose its specialty. See the below example for a better understanding.

#### **Example:**

The first search **(re.search(r'.', s))** matches any character, not just the period, while the second search **(re.search(r'\.', s))** specifically looks for and matches the period character.

* Python3

|  |
| --- |
| **import** re    s **=** 'geeks.forgeeks'    # without using \  match **=** re.search(r'.', s)  print(match)    # using \  match **=** re.search(r'\.', s)  print(match) |

**Output**

<re.Match object; span=(0, 1), match='g'>

<re.Match object; span=(5, 6), match='.'>

### 2. [] – Square Brackets

Square Brackets ([]) represent a character class consisting of a set of characters that we wish to match. For example, the character class [abc] will match any single a, b, or c.

We can also specify a range of characters using – inside the square brackets. For example,

* [0, 3] is sample as [0123]
* [a-c] is same as [abc]

We can also invert the character class using the caret(^) symbol. For example,

* [^0-3] means any number except 0, 1, 2, or 3
* [^a-c] means any character except a, b, or c

#### **Example:**

In this code, you’re using regular expressions to find all the characters in the string that fall within the range of ‘a’ to ‘m’. The **re.findall()** function returns a list of all such characters. In the given string, the characters that match this pattern are: ‘c’, ‘k’, ‘b’, ‘f’, ‘j’, ‘e’, ‘h’, ‘l’, ‘d’, ‘g’.

* Python3

|  |
| --- |
| **import** re    string **=** "The quick brown fox jumps over the lazy dog"  pattern **=** "[a-m]"  result **=** re.findall(pattern, string)    print(result) |

**Output**

['h', 'e', 'i', 'c', 'k', 'b', 'f', 'j', 'm', 'e', 'h', 'e', 'l', 'a', 'd', 'g']

### 3. ^ – Caret

Caret (^) symbol matches the beginning of the string i.e. checks whether the string starts with the given character(s) or not. For example –

* ^g will check if the string starts with g such as geeks, globe, girl, g, etc.
* ^ge will check if the string starts with ge such as geeks, geeksforgeeks, etc.

#### **Example:**

This code uses regular expressions to check if a list of strings starts with **“The”**. If a string begins with **“The,” it’s marked as “Matched”** otherwise, it’s labeled as**“Not matched”.**

* Python3

|  |
| --- |
| **import** re  regex **=** r'^The'  strings **=** ['The quick brown fox', 'The lazy dog', 'A quick brown fox']  **for** string **in** strings:  **if** re.match(regex, string):  **print**(f'Matched: {string}')  **else**:          print(f'Not matched: {string}') |

**Output**

Matched: The quick brown fox

Matched: The lazy dog

Not matched: A quick brown fox

### 4. $ – Dollar

Dollar($) symbol matches the end of the string i.e checks whether the string ends with the given character(s) or not. For example-

* s$ will check for the string that ends with a such as geeks, ends, s, etc.
* ks$ will check for the string that ends with ks such as geeks, geeksforgeeks, ks, etc.

#### **Example:**

This code uses a regular expression to check if the string ends with**“World!”.** If a match is found, it prints **“Match found!”** otherwise, it prints **“Match not found”**.

* Python3

|  |
| --- |
| **import** re    string **=** "Hello World!"  pattern **=** r"World!$"    match **=** re.search(pattern, string)  **if** match:      print("Match found!")  **else**:      print("Match not found.") |

**Output**

Match found!

### 5. . – Dot

Dot(.) symbol matches only a single character except for the newline character (\n). For example –

* a.b will check for the string that contains any character at the place of the dot such as acb, acbd, abbb, etc
* .. will check if the string contains at least 2 characters

#### **Example:**

This code uses a regular expression to search for the pattern **“brown.fox”**within the string. The dot (.) in the pattern represents any character. If a match is found, it prints **“Match found!”** otherwise, it prints **“Match not found”**.

* Python3

|  |
| --- |
| **import** re    string **=** "The quick brown fox jumps over the lazy dog."  pattern **=** r"brown.fox"    match **=** re.search(pattern, string)  **if** match:      print("Match found!")  **else**:      print("Match not found.") |

**Output**

Match found!

### 6. | – Or

Or symbol works as the or operator meaning it checks whether the pattern before or after the or symbol is present in the string or not. For example –

* a|b will match any string that contains a or b such as acd, bcd, abcd, etc.

### 7. ? – Question Mark

The question mark (?) is a quantifier in regular expressions that indicates that the preceding element should be matched zero or one time. It allows you to specify that the element is optional, meaning it may occur once or not at all. For example,

* ab?c will be matched for the string ac, acb, dabc but will not be matched for abbc because there are two b. Similarly, it will not be matched for abdc because b is not followed by c.

### 8.\* – Star

Star (\*) symbol matches zero or more occurrences of the regex preceding the \* symbol. For example –

* ab\*c will be matched for the string ac, abc, abbbc, dabc, etc. but will not be matched for abdc because b is not followed by c.

### 9. + – Plus

Plus (+) symbol matches one or more occurrences of the regex preceding the + symbol. For example –

* ab+c will be matched for the string abc, abbc, dabc, but will not be matched for ac, abdc, because there is no b in ac and b, is not followed by c in abdc.

### 10. {m, n} – Braces

Braces match any repetitions preceding regex from m to n both inclusive. For example –

* a{2, 4} will be matched for the string aaab, baaaac, gaad, but will not be matched for strings like abc, bc because there is only one a or no a in both the cases.

### 11. (<regex>) – Group

Group symbol is used to group sub-patterns. For example –

* (a|b)cd will match for strings like acd, abcd, gacd, etc.

## Special Sequences

Special sequences do not match for the actual character in the string instead it tells the specific location in the search string where the match must occur. It makes it easier to write commonly used patterns.

### List of special sequences

| **Special Sequence** | **Description** | **Examples** | |
| --- | --- | --- | --- |
| \A | Matches if the string begins with the given character | \Afor | for geeks |
| for the world |
| \b | Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word. | \bge | geeks |
| get |
| \B | It is the opposite of the \b i.e. the string should not start or end with the given regex. | \Bge | together |
| forge |
| \d | Matches any decimal digit, this is equivalent to the set class [0-9] | \d | 123 |
| gee1 |
| \D | Matches any non-digit character, this is equivalent to the set class [^0-9] | \D | geeks |
| geek1 |
| \s | Matches any whitespace character. | \s | gee ks |
| a bc a |
| \S | Matches any non-whitespace character | \S | a bd |
| abcd |
| \w | Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9\_]. | \w | 123 |
| geeKs4 |
| \W | Matches any non-alphanumeric character. | \W | >$ |
| gee<> |
| \Z | Matches if the string ends with the given regex | ab\Z | abcdab |
| abababab |

## RegEx Functions

**re**module contains many [functions](https://www.geeksforgeeks.org/python-functions/)that help us to search a string for a match.

Let’s see various functions provided by this module to work with regex in Python.

| **Function** | **Description** |
| --- | --- |
| re.findall() | finds and returns all matching occurrences in a list |
| re.compile() | Regular expressions are compiled into pattern objects |
| re.split() | Split string by the occurrences of a character or a pattern. |
| re.sub() | Replaces all occurrences of a character or patter with a replacement string. |
| re.escape() | Escapes special character |
| re.search() | Searches for first occurrence of character or pattern |

Let’s see the working of these RegEx functions with definition and examples:

## 1. re.findall()

Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.

**Finding all occurrences of a pattern**

This code uses a regular expression **(\d+)**to find all the sequences of one or more digits in the given string. It searches for numeric values and stores them in a list. In this example, it finds and prints the numbers**“123456789”** and **“987654321”**from the input string.

* Python3

|  |
| --- |
| **import** re  string **=** """Hello my Number is 123456789 and              my friend's number is 987654321"""  regex **=** '\d+'    match **=** re.findall(regex, string)  print(match) |

**Output**

['123456789', '987654321']

## ****2. re.compile()****

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

**Example 1:**

The code uses a regular expression pattern**[a-e]** to find and list all lowercase letters from ‘a’ to ‘e’ in the input string **“Aye, said Mr. Gibenson Stark”.** The output will be **['e', 'a', 'd', 'b', 'e']**, which are the matching characters.

* Python

|  |
| --- |
| **import** re  p **=** re.compile('[a-e]')    print(p.findall("Aye, said Mr. Gibenson Stark")) |

**Output**

['e', 'a', 'd', 'b', 'e', 'a']

**Understanding the Output:**

* First occurrence is ‘e’ in “Aye” and not ‘A’, as it is Case Sensitive.
* Next Occurrence is ‘a’ in “said”, then ‘d’ in “said”, followed by ‘b’ and ‘e’ in “Gibenson”, the Last ‘a’ matches with “Stark”.
* Metacharacter backslash ‘\’ has a very important role as it signals various sequences. If the backslash is to be used without its special meaning as metacharacter, use’\\’

**Example 2:**Set class [\s,.] will match any whitespace character,  ‘,’,  or, ‘.’ .

The code uses regular expressions to find and list all single digits and sequences of digits in the given input strings. It finds single digits with **\d**and sequences of digits with **\d+**.

* Python

|  |
| --- |
| **import** re  p **=** re.compile('\d')  print(p.findall("I went to him at 11 A.M. on 4th July 1886"))    p **=** re.compile('\d+')  print(p.findall("I went to him at 11 A.M. on 4th July 1886")) |

**Output**

['1', '1', '4', '1', '8', '8', '6']

['11', '4', '1886']

**Example 3:**

The code uses regular expressions to find and list word characters, sequences of word characters, and non-word characters in input strings. It provides lists of the matched characters or sequences.

* Python

|  |
| --- |
| **import** re    p **=** re.compile('\w')  **print**(p.findall("He said \* in some\_lang."))    p **=** re.compile('\w+')  **print**(p.findall("I went to him at 11 A.M., he \  said **\*\*\*** **in** some\_language."))    p **=** re.compile('\W')  print(p.findall("he said \*\*\* in some\_language.")) |

**Output**

['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '\_', 'l', 'a', 'n', 'g']

['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in', 'some\_language']

[' ', ' ', '\*', '\*', '\*', ' ...

**Example 4:**

The code uses a regular expression pattern ‘ab\*’ to find and list all occurrences of ‘ab’ followed by zero or more ‘b’ characters in the input string “ababbaabbb”. It returns the following list of matches: [‘ab’, ‘abb’, ‘abbb’].

* Python

|  |
| --- |
| **import** re  p **=** re.compile('ab\*')  print(p.findall("ababbaabbb")) |

**Output**

['ab', 'abb', 'a', 'abbb']

**Understanding the Output:**

* Our RE is ab\*, which ‘a’ accompanied by any no. of ‘b’s, starting from 0.
* Output ‘ab’, is valid because of single ‘a’ accompanied by single ‘b’.
* Output ‘abb’, is valid because of single ‘a’ accompanied by 2 ‘b’.
* Output ‘a’, is valid because of single ‘a’ accompanied by 0 ‘b’.
* Output ‘abbb’, is valid because of single ‘a’ accompanied by 3 ‘b’.

## ****3. re.split()****

Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.

**Syntax :**

re.split(pattern, string, maxsplit=0, flags=0)

The First parameter, pattern denotes the regular expression, string is the given string in which pattern will be searched for and in which splitting occurs, maxsplit if not provided is considered to be zero ‘0’, and if any nonzero value is provided, then at most that many splits occur. If maxsplit = 1, then the string will split once only, resulting in a list of length 2. The flags are very useful and can help to shorten code, they are not necessary parameters, eg: flags = re.IGNORECASE, in this split, the case, i.e. the lowercase or the uppercase will be ignored.

**Example 1:**

Splits a string using non-word characters and spaces as delimiters, returning words: **['Words', 'words', 'Words']**. Considers apostrophes as non-word characters: **['Word', 's', 'words', 'Words']**. Splits using non-word characters and digits:**['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']**. Splits using digits as the delimiter: **['On ', 'th Jan ', ', at ', ':', ' AM']**.

* Python

|  |
| --- |
| **from** re **import** split    **print**(split('\W+', 'Words, words , Words'))  print(split('\W+', "Word's words Words"))  **print**(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))  print(split('\d+', 'On 12th Jan 2016, at 11:02 AM')) |

**Output**

['Words', 'words', 'Words']

['Word', 's', 'words', 'Words']

['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']

['On ', 'th Jan ', ', at ', ':', ' AM']

**Example 2:**

First statement splits the string at the first occurrence of one or more digits: **['On ', 'th Jan 2016, at 11:02 AM']**. second splits the string using lowercase letters a to f as delimiters, case-insensitive:**['', 'y, ', 'oy oh ', 'oy, ', 'ome here'**]. Third splits the string using lowercase letters a to f as delimiters, case-sensitive: **['', 'ey, Boy oh ', 'oy, ', 'ome here']**.

* Python

|  |
| --- |
| **import** re  **print**(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))  **print**(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags**=**re.IGNORECASE))  print(re.split('[a-f]+', 'Aey, Boy oh boy, come here')) |

**Output**

['On ', 'th Jan 2016, at 11:02 AM']

['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']

['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']

## 4. ****re.sub()****

The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string(3rd parameter), and upon finding the substring pattern is replaced by repl(2nd parameter), count checks and maintains the number of times this occurs.

**Syntax:**

re.sub(pattern, repl, string, count=0, flags=0)

**Example 1:**

* First statement replaces all occurrences of ‘ub’ with ‘~\*’ (case-insensitive): **'S~\*ject has ~\*er booked already'**.
* Second statement replaces all occurrences of ‘ub’ with ‘~\*’ (case-sensitive): **'S~\*ject has Uber booked already'**.
* Third statement replaces the first occurrence of ‘ub’ with ‘~\*’ (case-insensitive): **'S~\*ject has Uber booked already'**.
* Fourth replaces ‘AND’ with ‘ & ‘ (case-insensitive): **'Baked Beans & Spam'.**
* Python

|  |
| --- |
| **import** re  print(re.sub('ub', '~\*', 'Subject has Uber booked already',               flags**=**re.IGNORECASE))  **print**(re.sub('ub', '~\*', 'Subject has Uber booked already'))  **print**(re.sub('ub', '~\*', 'Subject has Uber booked already',               count**=**1, flags**=**re.IGNORECASE))  print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',               flags**=**re.IGNORECASE)) |

**Output**

S~\*ject has ~\*er booked already

S~\*ject has Uber booked already

S~\*ject has Uber booked already

Baked Beans & Spam

## 5. ****re.subn()****

subn() is similar to sub() in all ways, except in its way of providing output. It returns a tuple with a count of the total of replacement and the new string rather than just the string.

**Syntax:**

re.subn(pattern, repl, string, count=0, flags=0)

**Example:**

**re.subn()** replaces all occurrences of a pattern in a string and returns a tuple with the modified string and the count of substitutions made. It’s useful for both case-sensitive and case-insensitive substitutions.

* Python

|  |
| --- |
| **import** re    **print**(re.subn('ub', '~\*', 'Subject has Uber booked already'))    t **=** re.subn('ub', '~\*', 'Subject has Uber booked already',              flags**=**re.IGNORECASE)  print(t)  print(len(t))  **print**(t[0]) |

**Output**

('S~\*ject has Uber booked already', 1)

('S~\*ject has ~\*er booked already', 2)

2

S~\*ject has ~\*er booked already

## 6. ****re.escape()****

Returns string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

**Syntax:**

re.escape(string)

**Example:**

**re.escape()** is used to escape special characters in a string, making it safe to be used as a pattern in regular expressions. It ensures that any characters with special meanings in regular expressions are treated as literal characters.

* Python

|  |
| --- |
| **import** re  print(re.escape("This is Awesome even 1 AM"))  **print**(re.escape("I Asked what is this [a-9], he said \t ^WoW")) |

**Output**

This\ is\ Awesome\ even\ 1\ AM

I\ Asked\ what\ is\ this\ \[a\-9\]\,\ he\ said\ \ \ \^WoW

## 7. re.search()

This method either returns None (if the pattern doesn’t match), or a re.MatchObject contains information about the matching part of the string. This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.

**Example:**Searching for an occurrence of the pattern

This code uses a regular expression to search for a pattern in the given string. If a match is found, it extracts and prints the matched portions of the string.

In this specific example, it searches for a pattern that consists of a month (letters) followed by a day (digits) in the input string “I was born on June 24”. If a match is found, it prints the full match, the month, and the day.

* Python3

|  |
| --- |
| **import** re  regex **=** r"([a-zA-Z]+) (\d+)"    match **=** re.search(regex, "I was born on June 24")  **if** match !**=** None:      print ("Match at index %s, %s" **%** (match.start(), match.end()))  **print** ("Full match: %s" **%** (match.group(0)))  **print** ("Month: %s" **%** (match.group(1)))  **print** ("Day: %s" **%** (match.group(2)))    **else**:      print ("The regex pattern does not match.") |

**Output**

Match at index 14, 21

Full match: June 24

Month: June

Day: 24

## SETS

A**Set** is a set of characters enclosed in ‘[]’ brackets. Sets are used to match a single character in the set of characters specified between brackets. Below is the list of Sets:

| **Set** | **Description** |
| --- | --- |
| \{n,\} | Quantifies the preceding character or group and matches at least n occurrences. |
| \* | Quantifies the preceding character or group and matches zero or more occurrences. |
| [0123] | Matches the specified digits (0, 1, 2, or 3) |
| [^arn] | matches for any character EXCEPT a, r, and n |
| \d | Matches any digit (0-9). |
| [0-5][0-9] | matches for any two-digit numbers from 00 and 59 |
| \w | Matches any alphanumeric character (a-z, A-Z, 0-9, or \_). |
| [a-n] | Matches any lower case alphabet between a and n. |
| \D | Matches any non-digit character. |
| [arn] | matches where one of the specified characters (a, r, or n) are present |
| [a-zA-Z] | matches any character between a and z, lower case OR upper case |
| [0-9] | matches any digit between 0 and 9 |

## Match Object

A Match object contains all the information about the search and the result and if there is no match found then None will be returned. Let’s see some of the commonly used methods and attributes of the match object.

### 1. Getting the string and the regex

**match.re** attribute returns the regular expression passed and **match.string** attribute returns the string passed.

**Example:**Getting the string and the regex of the matched object

The code searches for the letter **“G”** at a word boundary in the string **“Welcome to GeeksForGeeks”** and prints the regular expression pattern**(res.re)**and the original string **(res.string)**.

* Python3

|  |
| --- |
| **import** re  s **=** "Welcome to GeeksForGeeks"  res **=** re.search(r"\bG", s)    print(res.re)  print(res.string) |

**Output**

re.compile('\\bG')

Welcome to GeeksForGeeks

### 2. Getting index of matched object

* start() method returns the starting index of the matched substring
* end() method returns the ending index of the matched substring
* span() method returns a tuple containing the starting and the ending index of the matched substring

**Example:**Getting index of matched object

The code searches for the substring “Gee” at a word boundary in the string “Welcome to GeeksForGeeks” and prints the start index of the match (res.start()), the end index of the match (res.end()), and the span of the match (res.span()).

* Python3

|  |
| --- |
| **import** re    s **=** "Welcome to GeeksForGeeks"    res **=** re.search(r"\bGee", s)    print(res.start())  print(res.end())  print(res.span()) |

**Output**

11

14

(11, 14)

### 3. Getting matched substring

group() method returns the part of the string for which the patterns match. See the below example for a better understanding.

**Example:**Getting matched substring

The code searches for a sequence of two non-digit characters followed by a space and the letter ‘t’ in the string “Welcome to GeeksForGeeks” and prints the matched text using **res.group()**.

* Python3

|  |
| --- |
| **import** re  s **=** "Welcome to GeeksForGeeks"  res **=** re.search(r"\D{2} t", s)  print(res.group()) |

**Output**

me t

In the above example, our pattern specifies for the string that contains at least 2 characters which are followed by a space, and that space is followed by a t.

***VERBOSE in Python***

In Python regular expressions, "Verbose" mode allows you to write regular expressions in a format that's easier to read and understand. Normally, regular expressions can become quite complex and hard to read, especially when they're long or involve a lot of special characters.

When you use verbose mode (**re.VERBOSE** flag), you can add comments and extra whitespace to your regular expressions without affecting their functionality. This means you can break up your regular expression into multiple lines, add comments to explain what different parts of it do, and use whitespace to make it more visually organized.

For example, instead of writing a long, hard-to-read regular expression like this:

r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

You could use verbose mode to make it more readable like this:

r''' \b # start of word boundary [A-Za-z0-9.\_%+-]+ # username @ # @ symbol [A-Za-z0-9.-]+ # domain name \. # dot [A-Za-z]{2,} # top-level domain (2 or more characters) \b # end of word boundary '''

In this example, the **\b**, **@**, and **\.** are parts of the regular expression, while the comments help explain what each part does. The extra whitespace and line breaks make it easier to see the structure of the regular expression.

Using verbose mode can make your regular expressions easier to write, read, and maintain, especially for complex patterns.

***Indentation in Python***

- \*\*Definition\*\*:

- \*\*Importance\*\*: Crucial for Python code compilation and preventing IndentationError.

- \*\*Concept\*\*: Adding whitespace before a statement to define a block of code.

- \*\*Example\*\*:

- Statements with the same indentation level belong to the same block.

- Nested structures: Each level of indentation signifies a deeper block.

- \*\*Usage\*\*:

- Indicates code grouping and structure to the Python interpreter.

- Replaces braces used in other languages for block delineation.

- \*\*Syntax\*\*:

- Default: 4 spaces for each level of indentation.

- Customizable: Number of spaces can vary, but at least 1 space is required.

- \*\*Code Samples\*\*:

- Example 1:

site = 'gfg'

if site == 'gfg':

print('Logging on to geeksforgeeks...')

else:

print('retype the URL.')

print('All set !')

- Example 2:

j = 1

while j <= 5:

print(j)

j += 1

- \*\*Conclusion\*\*:

- Indentation is a fundamental aspect of Python syntax, delineating code blocks.

- It enhances code readability and maintains consistency, facilitating error-free execution.

